Contents

1 Polynomial Time **3**

2 Polynomial Time Verification **3**

3 Class NP **3**

4 Reducibility **4**

5 NP-Complete Problems **4**

6 How to Prove a Problem is NP-Complete **4**

7 Common NP-Complete Problems **5**
7.1 **Steps to Solve an NP Problem** 5

8 Summary Table **6**

9 Key Exam Points **7**

10 Two Circle Intersection **7**
10.1 Cases 7
10.2 Python Implementation 8
10.3 Illustrations 9

11 Check if a point is inside or outside of a polygon **10**

12 Quardic hashing proof **10**

13 Uniqueness Part of the Chinese Remainder Theorem **11**

14 28 Mid **12**

15 qs 1 **12**

16 27 Mid **14**

17 Final 27 **16**

18 Qs 1 **16**
18.1 qs 2 17

19 Qs 3 **18**
19.1 Qs 4 22

20 final 26 **22**

21 qs 1 **22**

22 batch 25 **23**

23 batch 24 **26**

24 qs 5 **27**

25 batch 23 **29**

1 Polynomial Time

Definition: An algorithm is said to run in **polynomial time** if its running time is

$$O(n^k)$$

for some constant k, where n is the size of the input.

Examples:

- Sorting: $O(n \log n)$
- Matrix multiplication: $O(n^3)$
- Graph BFS/DFS: $O(V + E)$

Class P: The class **P** consists of all decision problems that can be solved in polynomial time by a deterministic Turing machine.

$$\mathbf{P} = \{L \mid L \text{ is solvable in polynomial time}\}$$

—

2 Polynomial Time Verification

Definition: A problem has **polynomial-time verification** if, given a proposed solution (certificate), we can verify its correctness in polynomial time.

Example: Hamiltonian Cycle

- Certificate: A sequence of vertices
- Verification: Check if every vertex appears once and edges exist
- Time: Polynomial

Key Idea:

"Checking a solution is easier than finding it."

—

3 Class NP

Definition: The class **NP** consists of all decision problems for which a given solution can be verified in polynomial time.

$$\mathbf{NP} = \{L \mid L \text{ has polynomial-time verification}\}$$

Equivalent Definition:

- Solvable by a nondeterministic Turing machine in polynomial time

Relationship:

$$\mathbf{P} \subseteq \mathbf{NP}$$

Whether $\mathbf{P} = \mathbf{NP}$ is an **open problem**.

—

4 Reducibility

Polynomial-Time Reduction (Many-One Reduction):
A problem A is polynomial-time reducible to problem B, written

$$A \leq_p B$$

if any instance of A can be transformed into an instance of B in polynomial time such that:

$$A \text{ is YES} \iff B \text{ is YES}$$

Purpose of Reduction:

- Compare difficulty of problems
- Prove NP-completeness

Important Property:

If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.

—

5 NP-Complete Problems

Definition: A problem L is **NP-complete** if:

(i) $L \in \mathbf{NP}$

(ii) Every problem in **NP** is reducible to L in polynomial time

$$\mathbf{NPC} = \{L \mid L \in \mathbf{NP} \text{ and NP-hard}\}$$

Meaning:

- Hardest problems in NP
- If any NP-complete problem is in P, then $P = NP$

—

6 How to Prove a Problem is NP-Complete

1. Show the problem is in NP
2. Choose a known NP-complete problem
3. Reduce the known problem to the given problem in polynomial time

$$\text{Known NPC} \leq_p \text{New Problem}$$

—

7 Common NP-Complete Problems

- SAT (Boolean Satisfiability Problem)
- 3-SAT
- Clique
- Vertex Cover
- Hamiltonian Cycle
- Traveling Salesman Problem (Decision version)
- Subset Sum
- Knapsack (Decision version)

Cook–Levin Theorem: SAT was the first problem proven to be NP-complete.

7.1 Steps to Solve an NP Problem

NP (Nondeterministic Polynomial-time) problems are combinatorial problems for which no known polynomial-time algorithm exists. Examples include the Travelling Salesman Problem (TSP), Knapsack Problem, Graph Coloring, and Job Scheduling.

Steps:

1. **Problem Formulation:** Clearly define the input, output, and constraints.
   Example: In TSP, the input is a distance matrix, and the output is the shortest tour visiting all cities exactly once.
2. **Solution Representation:** Decide how a solution will be represented.
   Example: For Knapsack, a binary vector can represent item inclusion (‘1‘ for included, ‘0‘ for excluded).
3. **State Space Definition:** Identify all possible partial and complete solutions, often represented as a *state space tree*.
   Example: In N-Queens, each level of the tree corresponds to placing a queen in a row.
4. **Choose a Solution Approach:** Select an appropriate method:
   - **Exact Methods:**
     - Backtracking — explore all feasible solutions and prune infeasible ones.
     - Branch and Bound — use bounds to prune subtrees that cannot yield better solutions.
     - Dynamic Programming — break the problem into overlapping subproblems.
   - **Heuristic/Approximation Methods:** Used when exact solutions are too expensive.
     - Greedy algorithms, Genetic algorithms, Simulated Annealing, etc.

5. **Feasibility Check:** Ensure candidate solutions satisfy all constraints (e.g., weight limit in Knapsack, valid coloring in Graph Coloring).

6. **Optimization/Selection:** Among all feasible solutions, select the one that optimizes the objective function (maximize profit, minimize cost, etc.).

7. **Termination:** Stop when all possibilities have been explored (exact methods) or when a satisfactory solution is found (heuristic methods).

8 Summary Table

Class	Meaning	Example
P	Problems that can be solved in polynomial time	Sorting, Shortest path (BFS)
NP	Problems whose solutions can be verified in polynomial time	Hamiltonian Cycle (verification)
NP-Hard	Problems at least as hard as NP problems, not necessarily in NP	Optimization version of TSP
NP-Complete	Problems that are in NP and NP-Hard	SAT, 3-SAT, Vertex Cover

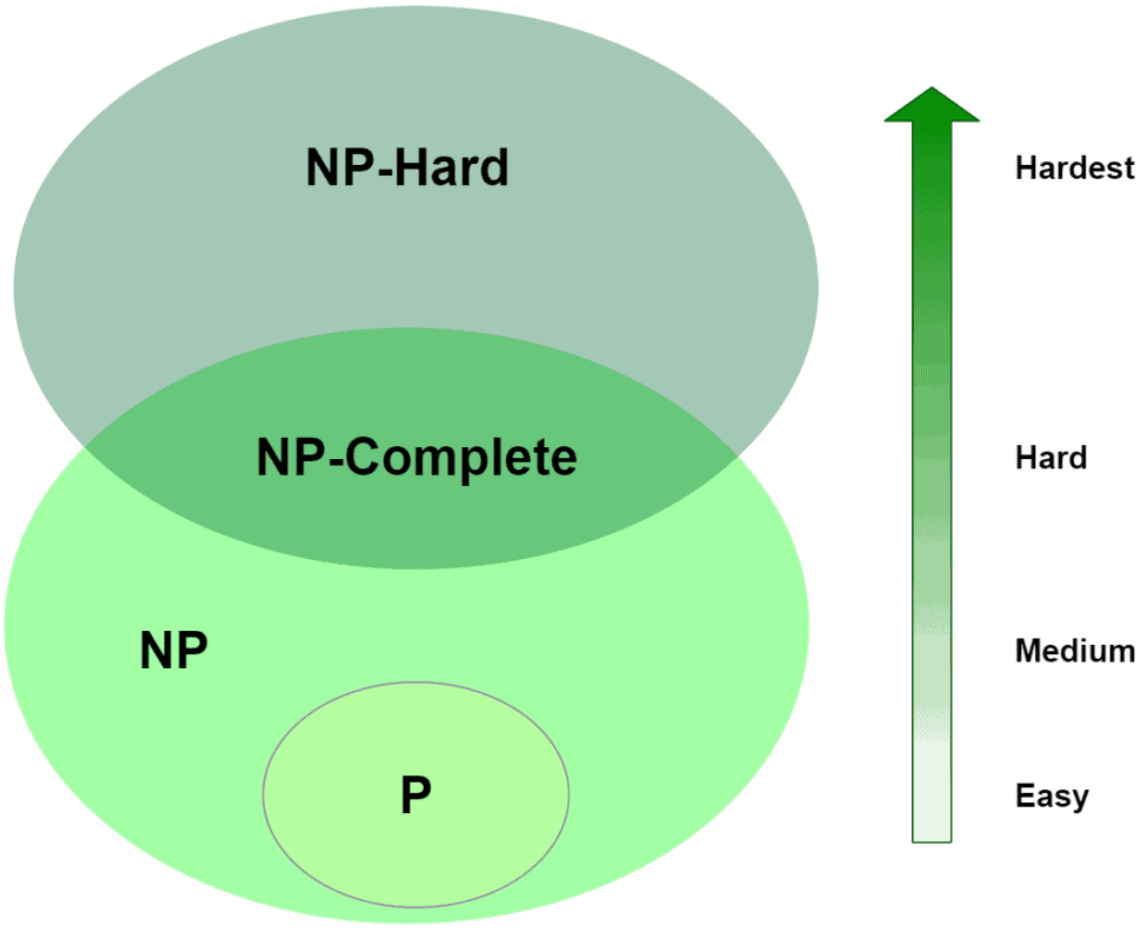


Figure 1: Caption

9 Key Exam Points

- P vs NP is unknown
- NP-complete problems are decision problems
- Reduction direction is very important
- Verification time defines NP, not solving time

Preorder: Process node → Left → Right
Inorder: Left → Node → Right
Postorder: Left → Right → Node

10 Two Circle Intersection

The distance between the centers $C_1(x_1, y_1)$ and $C_2(x_2, y_2)$ is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

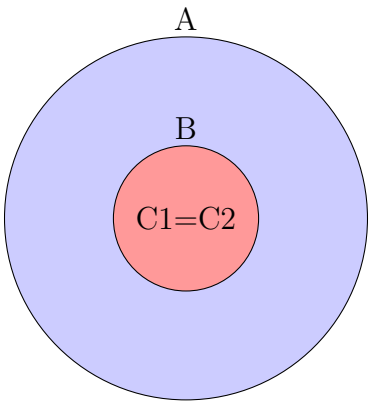
10.1 Cases

If $d \leq R_1 - R_2$	$\Rightarrow$	Circle B is inside A.
If $d \leq R_2 - R_1$	$\Rightarrow$	Circle A is inside B.
If $d < R_1 + R_2$	$\Rightarrow$	Circles intersect.
If $d = R_1 + R_2$	$\Rightarrow$	Circles touch externally.
If $d > R_1 + R_2$	$\Rightarrow$	Circles do not overlap.

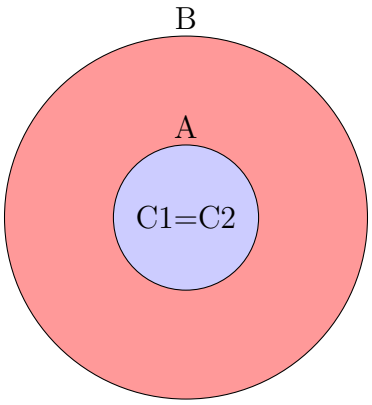
10.2 Python Implementation

```python
import math

# Function to check the relation between two circles
def circle_relation(x1, y1, r1, x2, y2, r2):
    """
    Determines the spatial relationship between two circles.
    """
    d = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

    if d <= abs(r1 - r2):
        if r1 > r2:
            print("Circle B is inside Circle A")
        elif r2 > r1:
            print("Circle A is inside Circle B")
        else:
            print("Circles are coincident (identical)")
    elif d < r1 + r2:
        print("Circles intersect each other")
    elif d == r1 + r2:
        print("Circles touch each other externally")
    else:
        print("Circles do not overlap")

# Example usage
if __name__ == "__main__":
    circle_relation(-10, 8, 30, 14, -24, 10)
```

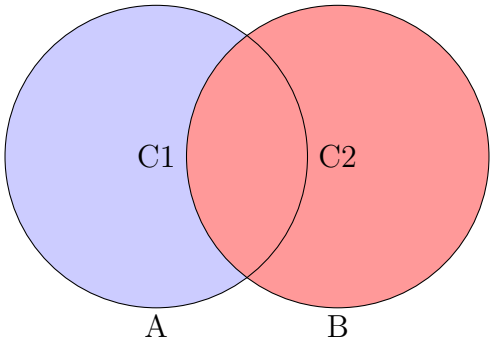
10.3 Illustrations



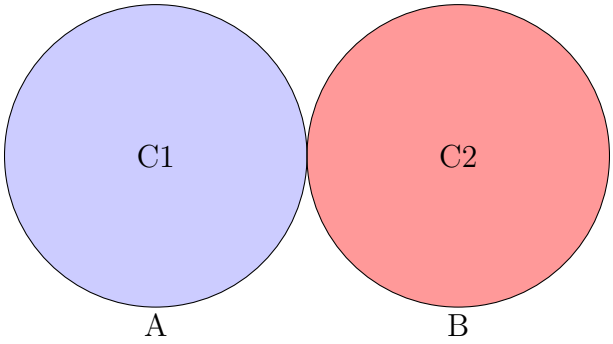
Case 1: Circle B inside A ($d \leq R_1 - R_2$).



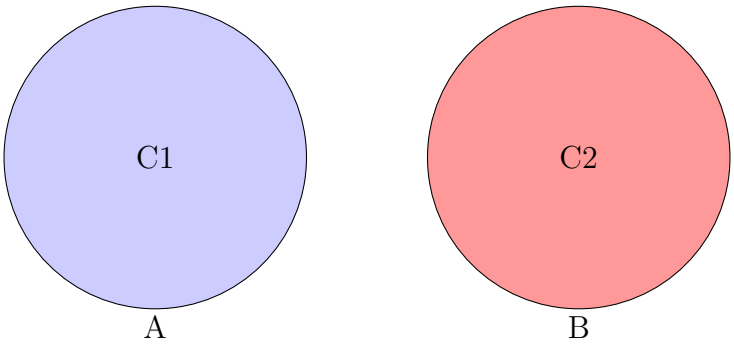
Case 2: Circle A inside B ($d \leq R_2 - R_1$).



Case 3: Circles intersect ($d < R_1 + R_2$).



Case 4: Circles touch externally ($d = R_1 + R_2$).



Case 5: Circles do not overlap ($d > R_1 + R_2$).

11 Check if a point is inside or outside of a polygon

Click here to visit Example click

12 Quardic hashing proof

Proof. Let m be the size of the hash table, and assume m is a prime number. The quadratic probing sequence for a key with a hash value $h'(k)$ is given by:

$$h(k, i) = (h'(k) + i^2) \pmod{m} \quad \text{for } i = 0, 1, 2, \ldots$$

We need to prove that if the load factor $\alpha \leq 0.5$, then an empty slot will always be found. This is guaranteed if the first $\lfloor m/2 \rfloor$ probes visit unique positions. The load factor $\alpha = \frac{N}{m}$, where N is the number of occupied slots. If $\alpha \leq 0.5$, then $\frac{N}{m} \leq 0.5$, which implies $N \leq \lfloor m/2 \rfloor$. The number of empty slots is $m - N \geq m - \lfloor m/2 \rfloor = \lceil m/2 \rceil$.

We will now prove by contradiction that the first $\lfloor m/2 \rfloor$ probes are unique. Assume, for contradiction, that two different probe numbers, i and j, where $0 \leq i < j \leq \lfloor m/2 \rfloor$, map to the same slot.

$$(h'(k) + i^2) \pmod{m} = (h'(k) + j^2) \pmod{m}$$

Subtracting $h'(k)$ from both sides, we get:

$$i^2 \pmod{m} = j^2 \pmod{m}$$

This can be rewritten as:

$$(j^2 - i^2) \pmod{m} = 0$$

Factoring the difference of squares gives:

$$(j - i)(j + i) \pmod{m} = 0$$

Since m is a prime number, by Euclid's lemma, if m divides a product of two numbers, it must divide at least one of them. Thus, either $(j-i) \pmod{m} = 0$ or $(j+i) \pmod{m} = 0$.

Case 1: $(j - i) \pmod{m} = 0$ This means $j - i$ is a multiple of m. However, we know that $0 \leq i < j \leq \lfloor m/2 \rfloor$, so:

$$0 < j - i \leq \lfloor m/2 \rfloor - 0 = \lfloor m/2 \rfloor$$

Since m is prime and $m \geq 2$, it follows that $\lfloor m/2 \rfloor < m$. Therefore, $0 < j - i < m$. A number in this range cannot be a multiple of m. This is a contradiction.

Case 2: $(j + i) \pmod{m} = 0$ This means $j + i$ is a multiple of m. We know that $0 \leq i < j \leq \lfloor m/2 \rfloor$, so:

$$0 < j + i \leq \lfloor m/2 \rfloor + \lfloor m/2 \rfloor = 2\lfloor m/2 \rfloor$$

For any prime $m > 2$, m is odd, so $m = 2k + 1$ for some integer k. Then $\lfloor m/2 \rfloor = k$.

$$j + i \leq 2\lfloor m/2 \rfloor = 2k < 2k + 1 = m$$

Therefore, $0 < j + i < m$. A number in this range cannot be a multiple of m. This is also a contradiction.

Since both cases lead to a contradiction, our initial assumption must be false. The first $\lfloor m/2 \rfloor$ probes must all land in unique slots. Because the load factor $\alpha \leq 0.5$, the number of occupied slots $N \leq \lfloor m/2 \rfloor$. The first $\lfloor m/2 \rfloor$ unique probes are guaranteed to check at least one empty slot. Therefore, an empty slot will always be found. □

13 Uniqueness Part of the Chinese Remainder Theorem

Theorem (Chinese Remainder Theorem – Uniqueness). Let $m_1, m_2, \ldots, m_k$ be positive integers such that

$$\gcd(m_1, m_2, \ldots, m_k) = 1.$$

Then the system of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

has *at most one* solution modulo $m_1 m_2 \cdots m_k$.

Proof

Let X_1 and X_2 be any two solutions of the above system of congruences. Then for each i with $1 \leq i \leq k$, we have

$$X_1 \equiv a_i \pmod{m_i} \quad \text{and} \quad X_2 \equiv a_i \pmod{m_i}.$$

Subtracting the two congruences gives

$$X_1 \equiv X_2 \pmod{m_i},$$

which implies

$$m_i \mid (X_1 - X_2).$$

Since this holds for every $i = 1, 2, \ldots, k$, it follows that

$$\text{lcm}(m_1, m_2, \ldots, m_k) \mid (X_1 - X_2).$$

Because $m_1, m_2, \ldots, m_k$ are pairwise relatively prime, their least common multiple is

$$\text{lcm}(m_1, m_2, \ldots, m_k) = m_1 m_2 \cdots m_k.$$

Therefore,

$$m_1 m_2 \cdots m_k \mid (X_1 - X_2),$$

which implies

$$X_1 \equiv X_2 \pmod{m_1 m_2 \cdots m_k}.$$

Hence, any two solutions of the system are congruent modulo $m_1 m_2 \cdots m_k$. This proves that the system of congruences has *at most one* solution modulo $m_1 m_2 \cdots m_k$.

□

14 28 Mid

15 qs 1

Index	0	1	2	3	4	5	6	7	8	9	10
Key	46	52	–	42	–	22	34	41	30	23	53

Q2. Hashing with Quadratic Probing

Given: Table size: $m = 4013$ (prime)

Hash function:

$$h(k) = (3k + 5) \bmod 4013$$

Quadratic probing:

$$h(k, i) = (h(k) + i^2) \bmod 4013$$

We need two consecutive probe indices i and $j = i + 1$, $i \neq j$, such that they produce the same table index.

Step 1: Condition for collision

$$h(k) + i^2 \equiv h(k) + (i + 1)^2 \pmod{4013}$$

Cancel $h(k)$:

$$i^2 \equiv (i+1)^2 \pmod{4013}$$

Step 2: Simplify

$$(i+1)^2 - i^2 = 2i + 1$$

So we need:

$$2i + 1 \equiv 0 \pmod{4013}$$

Step 3: Solve

$$2i \equiv -1 \equiv 4012 \pmod{4013}$$

Since 4013 is prime, we can divide by 2:

$$i \equiv \frac{4012}{2} = 2006$$

$$j = i + 1 = 2007$$

$$\boxed{i = 2006,\ j = 2007}$$

These consecutive probe indices produce the same table index.

—

Q3. Convex Hull Area After Adding an Interior Point

Given: Finite set of points P, area of convex hull A.

Pick any two points $p_i, p_j \in P$, and define:

$$q = \lambda p_i + (1-\lambda)p_j, \quad 0 < \lambda < 1$$

Let:

$$P' = P \cup \{q\}, \quad B = \text{area of convex hull of } P'$$

Key Insight: q lies on the line segment between p_i and p_j, so q is inside or on the boundary of the convex hull of P.

Adding an interior/boundary point does not expand the convex hull.

$$\boxed{A = B \quad \text{(or equivalently, } A \leq B\text{)}}$$

—

Q4. Rabin–Karp and Maximum Alphabet Size

Given: Integer size = 32 bits, maximum pattern length $m = 5$, want linear-time Rabin–Karp.

Step 1: Rabin–Karp constraint

$$|\Sigma|^m \leq 2^{32}, \quad m = 5$$

Step 2: Solve inequality

$$|\Sigma|^5 \leq 2^{32} \implies |\Sigma| \leq \sqrt[5]{2^{32}} = 2^{32/5} \approx 84$$

$$\boxed{|\Sigma|_{\max} = 84}$$

16 27 Mid

Hash Table Insertion Problem

Given:

- Hash table length: $m = 11$
- Keys to insert: 10, 22, 31, 4, 15, 28, 17, 88, 59
- Auxiliary hash function: $h'(k) = k$

Method 1: Quadratic Probing with $C_1 = 1$ and $C_2 = 3$

Hash function:

$$h(k, i) = (h'(k) + C_1 \cdot i + C_2 \cdot i^2) \bmod m = (k + i + 3i^2) \bmod 11$$

Insertion Process:

1. **Key 10:** $h(10, 0) = 10 \bmod 11 = 10$ ✓
2. **Key 22:** $h(22, 0) = 22 \bmod 11 = 0$ ✓
3. **Key 31:** $h(31, 0) = 31 \bmod 11 = 9$ ✓
4. **Key 4:** $h(4, 0) = 4 \bmod 11 = 4$ ✓
5. **Key 15:**

   $h(15, 0) = 15 \bmod 11 = 4$ (occupied)
   $h(15, 1) = (15 + 1 + 3) \bmod 11 = 19 \bmod 11 = 8$ ✓
6. **Key 28:** $h(28, 0) = 28 \bmod 11 = 6$ ✓
7. **Key 17:**

   $h(17, 0) = 17 \bmod 11 = 6$ (occupied)
   $h(17, 1) = (17 + 1 + 3) \bmod 11 = 21 \bmod 11 = 10$ (occupied)
   $h(17, 2) = (17 + 2 + 12) \bmod 11 = 31 \bmod 11 = 9$ (occupied)
   $h(17, 3) = (17 + 3 + 27) \bmod 11 = 47 \bmod 11 = 3$ ✓
8. **Key 88:**

   $h(88, 0) = 88 \bmod 11 = 0$ (occupied)
   $h(88, 1) = (88 + 1 + 3) \bmod 11 = 92 \bmod 11 = 4$ (occupied)
   $h(88, 2) = (88 + 2 + 12) \bmod 11 = 102 \bmod 11 = 3$ (occupied)
   $h(88, 3) = (88 + 3 + 27) \bmod 11 = 118 \bmod 11 = 8$ (occupied)
   $h(88, 4) = (88 + 4 + 48) \bmod 11 = 140 \bmod 11 = 8$ (cycle)

Key 88 cannot be inserted due to cycling.

Final Hash Table (Quadratic Probing):

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	-	-	17	4	-	28	-	15	31	10

Method 2: Double Hashing

Hash functions:

$$h_1(k) = k \bmod 11$$
$$h_2(k) = 1 + (k \bmod (m-1)) = 1 + (k \bmod 10)$$
$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod 11$$

Insertion Process:

1. **Key 10:** $h_1(10) = 10$, position $= 10$ ✓
2. **Key 22:** $h_1(22) = 0$, position $= 0$ ✓
3. **Key 31:** $h_1(31) = 9$, position $= 9$ ✓
4. **Key 4:** $h_1(4) = 4$, position $= 4$ ✓
5. **Key 15:** $h_1(15) = 4$ (occupied), $h_2(15) = 1 + 5 = 6$

   $h(15,1) = (4+6) \bmod 11 = 10$ (occupied)
   $h(15,2) = (4+12) \bmod 11 = 5$ ✓
6. **Key 28:** $h_1(28) = 6$, position $= 6$ ✓
7. **Key 17:** $h_1(17) = 6$ (occupied), $h_2(17) = 1 + 7 = 8$

   $h(17,1) = (6+8) \bmod 11 = 3$ ✓
8. **Key 88:** $h_1(88) = 0$ (occupied), $h_2(88) = 1 + 8 = 9$

   $h(88,1) = (0+9) \bmod 11 = 9$ (occupied)
   $h(88,2) = (0+18) \bmod 11 = 7$ ✓
9. **Key 59:** $h_1(59) = 4$ (occupied), $h_2(59) = 1 + 9 = 10$

   $h(59,1) = (4+10) \bmod 11 = 3$ (occupied)
   $h(59,2) = (4+20) \bmod 11 = 2$ ✓

Final Hash Table (Double Hashing):

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	-	59	17	4	15	28	88	-	31	10

17 Final 27

18 Qs 1

Primary Clustering:

Primary clustering is a phenomenon in open addressing hash tables (especially linear probing) where a group of consecutive occupied slots forms a cluster. Once a cluster is formed, new keys hashing to any position within or near the cluster will probe through the entire cluster, causing it to grow larger.

Effect of Primary Clustering:

- Increases the average number of probes for insertion and search.
- Performance degrades rapidly as the load factor increases.
- Causes long probe sequences due to contiguous occupied slots.

Mechanisms to Reduce Primary Clustering:

- **Quadratic Probing:** Uses quadratic offsets to spread probes and avoid contiguous clusters.
- **Double Hashing:** Uses a second hash function to generate probe steps, providing better distribution.
- **Lower Load Factor:** Keeping α small reduces clustering.

Quadratic Probing Insertion:

Key	$h'(k)$	Final Position
10	10	10
22	0	0
31	9	9
4	4	4
15	4	8
28	6	6
17	6	3
88	0	1
59	4	7

m

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	88	–	17	4	–	28	59	15	31	10

Double Hashing Insertion:

Key	$h_1(k)$	$h_2(k)$	Final Position
10	10	1	10
22	0	3	0
31	9	2	9
4	4	5	4
15	4	6	3
28	6	9	6
17	6	8	5
88	0	9	7
59	4	10	2

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	–	59	15	4	17	28	88	–	31	10

Proof. Consider an open-address hash table with m slots and n keys, where the load factor is $\alpha = \frac{n}{m} < 1$. Assume uniform hashing.

For an unsuccessful search, each probe hits an occupied slot with probability α and an empty slot with probability $1 - \alpha$.

Thus, the probability that the first $k-1$ probes hit occupied slots and the k-th probe finds an empty slot is:

$$\alpha^{k-1}(1-\alpha)$$

The expected number of probes is:

$$E = \sum_{k=1}^{\infty} k\alpha^{k-1}(1-\alpha)$$

Using the identity for a geometric distribution:

$$E = \frac{1}{1-\alpha}$$

Therefore, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$. □

18.1 qs 2

State Space

In backtracking, the *state space* is the set of all possible states (partial or complete solutions) that can be generated while solving a problem. Each state represents a configuration of decisions made so far. The state space is usually represented as a *state space tree*.

State Space Tree

- The root node represents the initial state where no decision has been made.
- Each internal node represents a partial solution.
- Each edge represents a choice or decision.
- Leaf nodes represent complete solutions (valid or invalid).

Features of State Space in Backtracking

1. **Tree Structure:** The state space is organized in the form of a tree called the state space tree.
2. **Root Node:** The root represents the initial state of the problem with no variables assigned.
3. **Levels of the Tree:** Each level corresponds to a decision or assignment of a variable.
4. **Partial Solutions:** Nodes at intermediate levels represent partial solutions constructed so far.

5. **Branching:** Each node can have multiple children, each corresponding to a different possible choice.
6. **Feasibility Checking:** A partial solution is checked against the problem constraints. If it violates any constraint, the corresponding subtree is pruned.
7. **Pruning:** Infeasible states are eliminated early to reduce the search space.
8. **Solution Nodes:** Nodes that satisfy all constraints represent valid solutions.
9. **Depth-First Search:** Backtracking explores the state space using depth-first traversal.
10. **Finite State Space:** The number of states is finite, ensuring termination of the algorithm.

Linear Time Algorithm for N-Queens

There is no general linear-time algorithm to find *all* solutions of the N-Queens problem. However, a valid solution can be *constructed* in $\mathcal{O}(n)$ time for all $n \geq 4$.

Represent the solution by an array $Q[1 \dots n]$, where $Q[i]$ denotes the column position of the queen in row i.

Algorithm:

- If $n = 1$, the solution is [1].
- If $n = 2$ or $n = 3$, no solution exists.
- If n is even, place queens in columns:

$$2, 4, 6, \dots, n, 1, 3, 5, \dots, n-1$$

- If n is odd ($n \geq 5$), apply the even case for $n-1$ and place the last queen at (n, n).

This construction avoids column and diagonal conflicts and runs in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ space.

19 Qs 3

Best-First Branch and Bound

Best-first branch and bound is a problem-solving technique used to find the optimal solution of optimization problems. It is an extension of the branch and bound method in which the next node to be expanded is chosen based on the *best (most promising) bound value* rather than following depth-first order.

A priority queue is used to select the live node with the best bound.

Working Principle

- The solution space is represented as a state space tree.
- Each node represents a partial solution.
- A bound (upper or lower) is computed for each node.

- The node with the best bound value is selected for expansion.
- Nodes whose bound is worse than the current best solution are pruned.

Algorithm Steps

1. Start with the root node.
2. Insert the root into a priority queue.
3. Select the node with the best bound.
4. If the node represents a complete solution, update the best solution.
5. Otherwise, branch the node to generate children.
6. Compute bounds for child nodes and insert promising nodes into the queue.
7. Repeat until the queue is empty.

Example: 0/1 Knapsack Problem
Consider a knapsack with capacity $W = 10$ and the following items:

Item	Weight	Profit
1	2	40
2	3	50
3	5	100

In best-first branch and bound:

- The root node represents selecting no items.
- Bounds are calculated using fractional knapsack.
- The node with the highest bound is expanded first.
- Nodes that cannot yield a better profit than the current best are pruned.

The algorithm explores the most promising combinations first and finds the optimal solution efficiently.

Advantages

- Finds optimal solution faster than depth-first branch and bound
- Reduces unnecessary exploration

Disadvantages

- Requires additional memory for priority queue
- Bound computation can be expensive

Best-Case Input for Assignment Problem using Branch and Bound

In the branch and bound method for the assignment problem, the *best case* occurs when an optimal assignment is found at the upper level of the state space tree and all other branches are pruned early due to poor bounds.

This typically happens when the cost matrix is already arranged such that the minimum-cost assignments lie along the main diagonal.

Example

Consider the following cost matrix for assigning 3 workers to 3 jobs:

	J_1	J_2	J_3
W_1	1	10	10
W_2	10	2	10
W_3	10	10	3

Explanation

- The minimum cost in each row occurs in a distinct column.
- Assigning $W_1 \to J_1$, $W_2 \to J_2$, and $W_3 \to J_3$ gives total cost:

$$1 + 2 + 3 = 6$$

- The lower bound at the root node equals the optimal solution.
- All other branches have higher bounds and are pruned immediately.

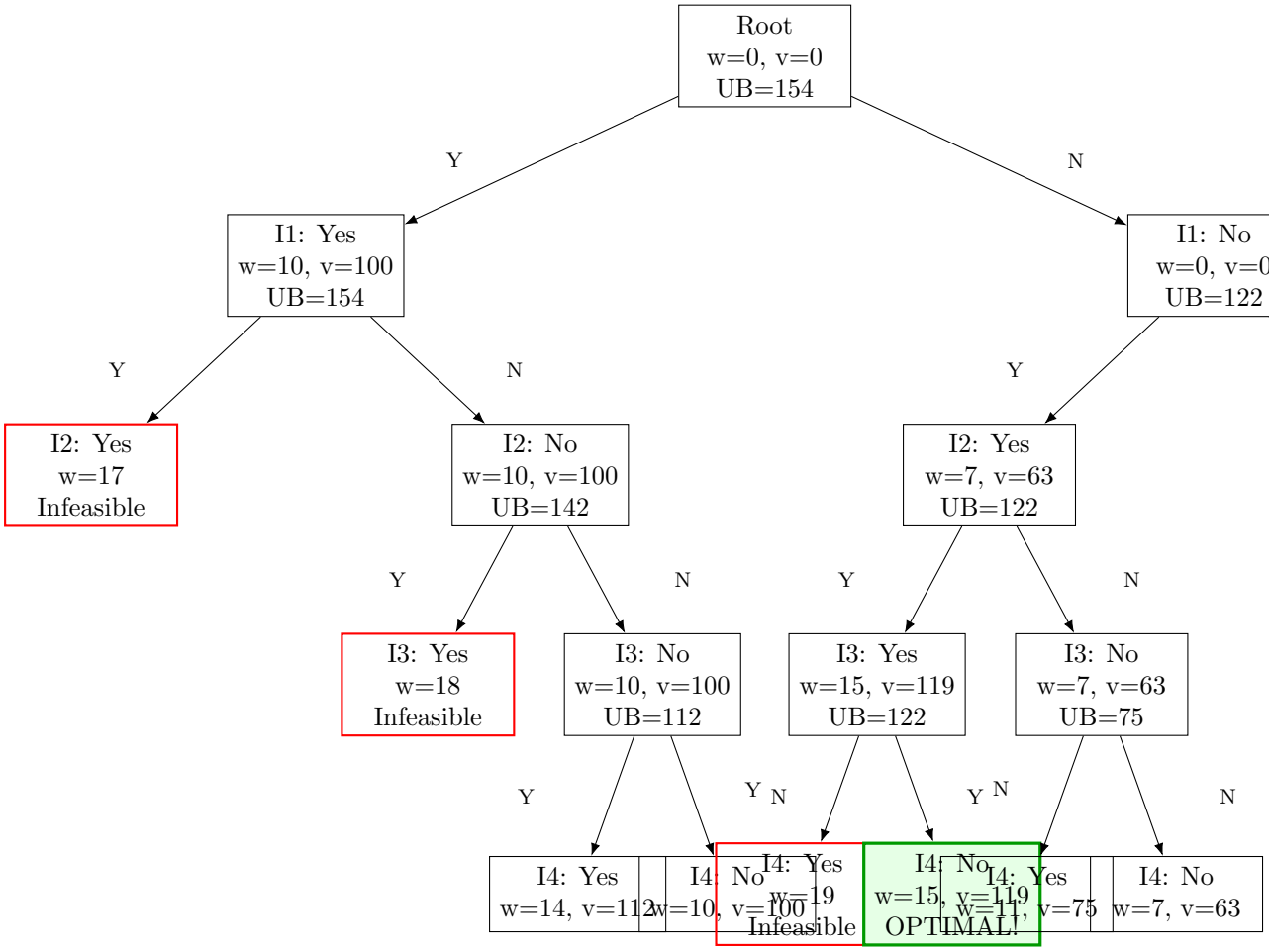
Thus, the algorithm explores very few nodes and quickly finds the optimal solution.

Conclusion

The best-case input for the assignment problem using branch and bound is a cost matrix where the optimal assignment is obvious and discovered early, allowing maximum pruning of the search space.

Branch and Bound Tree for Knapsack Problem

Problem: W = 16
Item 1: w=10, v=100, ratio=10.0
Item 2: w=7, v=63, ratio=9.0
Item 3: w=8, v=56, ratio=7.0
Item 4: w=4, v=12, ratio=3.0



Key Upper Bound Calculations:

- **Root:** $0 + 100 + (6/7) \times 63 = 154$
- **I1=Yes, I2=No:** $100 + (6/8) \times 56 = 100 + 42 = 142$
- **I1=Yes, I2=No, I3=No:** $100 + 12 = 112$
- **I1=No:** $0 + 63 + 56 + (1/4) \times 12 = 122$
- **I1=No, I2=Yes:** $63 + 56 + (1/4) \times 12 = 122$
- **I1=No, I2=Yes, I3=Yes:** $119 + (1/4) \times 12 = 122$

Optimal Solution: Select **Items 2 and 3**
Total Weight: $7 + 8 = 15 \leq 16$
Total Value: $63 + 56 =$ **119**

Note: The greedy approach (highest ratio first) gives Items 1+4 with value 112, but Branch and Bound explores all branches and finds the true optimal: Items 2+3 with value 119.

Step	i	j
1	0	0
2	1	1
3	2	2
4	2	0
5	3	1
6	3	0
7	4	1
8	4	0
9	5	1
10	6	2
11	7	3
12	8	4
13	9	5
14	10	6
15	10	2
16	11	3
17	11	0
18	12	0
19	13	1
20	14	2
21	14	0
22	15	1
23	16	2
24	16	0
25	17	1

Table 1: KMP Algorithm trace for text "abaaabbaabbbababa" and pattern "abbaab"

19.1 Qs 4

20 final 26

21 qs 1

Double Hashing Insertions:

Key	$h_1(k)$	$h_2(k)$	Probes	Final Slot
10	10	1	10	10
22	0	3	0	0
31	9	2	9	9
4	4	5	4	4
15	4	6	4, 10, 5	5
28	6	9	6	6
17	6	8	6, 3	3
88	0	9	0, 9, 7	7
59	4	10	4, 3, 2	2

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	–	59	17	4	15	28	88	–	31	10

Answer: Disagree.

For open addressing with uniform hashing:

- Expected probes for an **unsuccessful** search:

$$E_u = \frac{1}{1-\alpha}$$

- Expected probes for a **successful** search:

$$E_s \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Substituting $\alpha = 0.818$:

$$E_u = \frac{1}{1-0.818} \approx 5.49$$

Since the expected number of probes exceeds 4, the statement is **false**.

22 batch 25

Solution

6(a) Optimal Solution from Decision Problem

Yes, for many optimization problems, the optimal solution can be obtained using the corresponding decision problem.

Consider a minimization problem. The decision version asks whether there exists a solution of cost at most k. By performing a binary search on k and repeatedly solving the decision problem, the optimal value can be determined in polynomially many steps.

Hence, if the decision problem can be solved in polynomial time, the optimization problem can also be solved in polynomial time.

—

6(b) Suppose $Y \leq_p X$. Justify the following

(i) If X can be solved in polynomial time, then Y can be solved in polynomial time

Since $Y \leq_p X$, there exists a polynomial-time reduction from Y to X. Therefore, any instance of Y can be transformed into an instance of X in polynomial time.

If X has a polynomial-time algorithm, then we can:

1. Reduce Y to X in polynomial time.
2. Solve X in polynomial time.

Thus, Y can be solved in polynomial time.

(ii) If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time

Assume, for contradiction, that X can be solved in polynomial time. From part (i), this would imply that Y can also be solved in polynomial time, which contradicts the given assumption.

Hence, X cannot be solved in polynomial time.

—

6(c) Prove that the Traveling Salesman Problem is NP-Complete

Step 1: TSP is in NP

Given a tour, we can verify in polynomial time that:

- Each city is visited exactly once.
- The total tour cost is less than or equal to a given bound.

Hence, TSP belongs to NP.

Step 2: TSP is NP-Hard

We reduce the Hamiltonian Cycle problem to TSP. Given a graph $G(V, E)$, construct a complete graph where:

- Edge weight = 1 if the edge exists in G.
- Edge weight = 2 otherwise.

There exists a Hamiltonian cycle in G if and only if there exists a TSP tour of cost at most $|V|$.

Since Hamiltonian Cycle is NP-complete, TSP is NP-hard.

Conclusion: TSP is NP-complete.

—

6(d) Show that 3-CNF-SAT is reducible to Clique

Let the Boolean formula be in 3-CNF form with clauses:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

Construct a graph as follows:

- Create a vertex for each literal in each clause.
- Connect two vertices if they belong to different clauses and are not complementary.

The formula is satisfiable if and only if the graph contains a clique of size m.

Thus,

$$\text{3-CNF-SAT} \leq_p \text{CLIQUE}$$

—

7(a) Why RSA is trivial to crack if $\phi(n)$ is known

Let $n = pq$, where p and q are primes. Then:

$$\phi(n) = (p-1)(q-1) = pq - p - q + 1$$

Thus,

$$p + q = n - \phi(n) + 1$$

Knowing $p + q$ and pq, we can solve the quadratic equation:

$$x^2 - (p+q)x + pq = 0$$

This yields p and q. Once p and q are known, the private key can be computed easily, making RSA trivial to crack.

—

7(b) Solving the Modular Equation $ax \equiv b \pmod{n}$

Let $d = \gcd(a, n)$.

Condition for solvability: The equation has a solution if and only if $d \mid b$.

Finding the solution: Using the Extended Euclidean Algorithm, find integers x_0, y_0 such that:

$$ax_0 + ny_0 = d$$

Multiplying both sides by $\frac{b}{d}$ gives:

$$ax \equiv b \pmod{n}$$

There are exactly d distinct solutions modulo n.

—

7(c) What does EXTENDED-EUCLID(F_k, F_{k+1}) return?

Consecutive Fibonacci numbers are coprime:

$$\gcd(F_k, F_{k+1}) = 1$$

The Extended Euclidean Algorithm returns integers x, y such that:

$$F_k x + F_{k+1} y = 1$$

Using Fibonacci identities, the coefficients are:

$$x = (-1)^k, \quad y = (-1)^{k+1}$$

Thus, EXTENDED-EUCLID(F_k, F_{k+1}) returns $(1, (-1)^k, (-1)^{k+1})$.

23 batch 24

Question 2

2(a) Orientation using Cross Product

Let

$$p_1 = (x_1, y_1), \quad p_2 = (x_2, y_2)$$

be two vectors with respect to the origin $(0, 0)$.

The cross product (z-component) is defined as:

$$p_1 \times p_2 = x_1 y_2 - y_1 x_2$$

Case 1: If $p_1 \times p_2 > 0$

This means:

$$x_1 y_2 - y_1 x_2 > 0$$

Hence, vector p_1 lies in the clockwise direction from vector p_2 with respect to the origin.

Case 2: If $p_1 \times p_2 < 0$

Then:

$$x_1 y_2 - y_1 x_2 < 0$$

which implies that vector p_1 lies in the counterclockwise direction from vector p_2.

Conclusion: The sign of the cross product determines the relative orientation of two vectors.

—

2(b) Why Testing Only the x-Dimension is Incorrect

Professor Jami suggests that only the x-dimension needs to be tested to determine whether a point lies on a line segment.

This is incorrect because a point may satisfy the x-coordinate condition but fail the y-coordinate condition.

Example:

Consider the segment from $(0, 0)$ to $(2, 2)$ and the point $(1, 3)$.

Although 1 lies between 0 and 2 in the x-dimension, the point is not on the line $y = x$.

Conclusion: Both x- and y-dimensions must be tested to correctly determine whether a point lies on a line segment.

—

2(c) Intersection of a Right Horizontal Ray with a Line Segment

Given a point:

$$p_0 = (x_0, y_0)$$

The right horizontal ray from p_0 is:

$$\{(x, y) \mid x \geq x_0, y = y_0\}$$

To determine whether this ray intersects a line segment $p_1 p_2$ in $O(1)$ time, we reduce the problem to checking whether two line segments intersect.

Steps:

1. Extend the ray to a finite segment p_0p_3, where $p_3 = (M, y_0)$ and M is a sufficiently large value.
2. Check whether segments p_0p_3 and p_1p_2 intersect using orientation tests.

Since segment intersection testing is done in constant time, the ray-segment intersection can also be determined in $O(1)$ time.

—

2(d) Graham Scan Algorithm for Convex Hull

Pseudocode:

```
GRAHAM_SCAN(P):
    Find point p0 with lowest y-coordinate
    Sort remaining points by polar angle with p0
    Push p0, P[1], P[2] onto stack S

    for i = 3 to n:
        while orientation(next_to_top(S), top(S), P[i]) != CCW:
            pop(S)
        push(S, P[i])

    return S
```

—

Case (i): Including a Point in the Convex Hull

A point is included as a vertex of the convex hull if the turn formed with the previous two points is counterclockwise.

This indicates that the point contributes to the outer boundary of the point set.

—

Case (ii): Excluding a Point from the Convex Hull

A point is excluded if it forms a clockwise turn or is collinear with the previous points.

Such points lie inside the hull or on its edges and are removed during the scan.

—

Conclusion: Graham Scan efficiently constructs the convex hull by maintaining only points that form counterclockwise turns.

24 qs 5

Question 5

5(a) Circuit-Satisfiability is NP-Complete

Circuit-SAT is the problem of determining whether there exists an assignment to the input variables of a Boolean circuit such that the output is 1.

Step 1: Circuit-SAT $\in$ NP

Given a Boolean circuit and a truth assignment to its input variables, we can evaluate the circuit in polynomial time by computing the output gate by gate.

Hence, Circuit-SAT belongs to NP.

Step 2: Circuit-SAT is NP-Hard

The Boolean Satisfiability problem (SAT) is known to be NP-complete.

Any Boolean formula in CNF form can be transformed into an equivalent Boolean circuit using AND, OR, and NOT gates in polynomial time.

Thus,

$$\text{SAT} \leq_p \text{Circuit-SAT}$$

Conclusion:

Since Circuit-SAT is in NP and is NP-hard, it is NP-complete.

—

5(b) Vertex Cover is Polynomial-Time Reducible to TSP

Let $G = (V, E)$ be an undirected graph and k be a positive integer.

Vertex Cover Problem: Determine whether there exists a vertex cover of size at most k.

Construction:

Construct a complete weighted graph G' from G as follows:

- Each vertex in V becomes a city in G'.
- Assign weight 1 to edges present in E.
- Assign weight 2 to edges not present in E.

Set the bound for TSP as:

$$B = |V| + k$$

Correctness:

If G has a vertex cover of size at most k, then there exists a Hamiltonian tour in G' with total cost at most B.

Conversely, a tour with cost at most B implies the existence of a vertex cover of size at most k.

Conclusion:

Since the reduction can be done in polynomial time,

$$\text{Vertex Cover} \leq_p \text{TSP}$$

—

5(c) Compressed Trie Construction

Given text (case-insensitive):

"the bold bear became bored bearing the burden of its bland beauty"

Distinct Words:

$$\{\text{the}, \text{bold}, \text{bear}, \text{became}, \text{bored}, \text{bearing}, \text{burden}, \text{of}, \text{its}, \text{bland}, \text{beauty}\}$$

Compressed Trie

Common prefixes such as `be` and `bo` are merged into single edges.

Example compressed branches:

- `be` → `ar`, `came`, `aring`, `auty`
- `bo` → `ld`, `red`
- `bl` → `and`

Each node represents the longest possible common prefix.

—

Compact Array Representation

The compressed trie can be stored using an array of words:

$$[\text{the}, \text{bold}, \text{bear}, \text{became}, \text{bored}, \text{bearing}, \text{burden}, \text{of}, \text{its}, \text{bland}, \text{beauty}]$$

Each entry stores:

- The substring
- Pointer to next word fragment
- End-of-word marker

This representation significantly reduces space complexity compared to a standard trie.

—

Conclusion:
Compressed tries reduce both height and space while enabling efficient word matching.

25 batch 23

1 (a) (i) Distinct Permutations of a String with Duplicate Characters

Objective: Write a pseudocode to generate all **distinct permutations** of a given string containing duplicate characters with time complexity

$$O(n^2 \times n!)$$

Algorithm Idea

- Sort the string so that duplicate characters are adjacent.
- Use backtracking to build permutations.
- Use a boolean array to track used characters.
- Skip duplicate characters at the same recursion level.

Pseudocode

```
procedure DISTINCT_PERMUTATION(s)
    sort(s)
    n ← length(s)
    used[0..n-1] ← false
    backtrack("", used)
end procedure

procedure backtrack(current, used[])
    if length(current) = n then
        print current
        return
    end if

    for i ← 0 to n-1 do
        if used[i] = true then
            continue
        end if

        if i > 0 AND s[i] = s[i-1] AND used[i-1] = false then
            continue
        end if

        used[i] ← true
        backtrack(current + s[i], used)
        used[i] ← false
    end for
end procedure
```

Time Complexity Analysis

- Total permutations: $n!$
- Each permutation construction costs $O(n)$
- Loop overhead contributes another $O(n)$

$$\boxed{O(n^2 \times n!)}$$

—

1 (a) (ii) Permutations of a String without Duplicate Characters

Modification:

- Since all characters are unique, sorting and duplicate checking are removed.

Modified Pseudocode

```
procedure PERMUTATION(s)
    n ← length(s)
    used[0..n-1] ← false
    backtrack("", used)
end procedure

procedure backtrack(current, used[])
    if length(current) = n then
        print current
        return
    end if

    for i ← 0 to n-1 do
        if used[i] = false then
            used[i] ← true
            backtrack(current + s[i], used)
            used[i] ← false
        end if
    end for
end procedure
```

Time Complexity Proof

- Total permutations: $n!$
- Each recursive call performs $O(n)$ work

$$\boxed{O(n^2 \times n!)}$$

—

1 (b) Backtracking Simulation of N-Queen Problem (N = 3)

Approach

- Place one queen per row.
- Ensure no two queens share the same column or diagonal.

Simulation Steps

- Row 0: Try columns 0, 1, 2
- For each placement, attempt to place a queen in Row 1
- Continue until Row 2
- All configurations lead to conflict

Observation

- Every possible placement violates column or diagonal constraints.

Conclusion

No solution exists for the N-Queen problem when $N = 3$

Answer

1 (c) Searching All Occurrences of a Pattern and Its Permutations

Problem Statement: Given a text array `txt[0..n-1]` and a pattern array `pat[0..m-1]`, write a linear-time function `Search(char pat[], char txt[])` that prints all occurrences of `pat[]` and its permutations (anagrams) in `txt[]`. Assume that:

- $n > m$
- All characters in the pattern are distinct

Also prove that the time complexity is $O(n)$.

—

Algorithm Idea

- Use the **sliding window technique** of size m.
- Maintain frequency count arrays for pattern and current window.
- If the frequency arrays match, an anagram is found.

—

Pseudocode

```
procedure Search(pat, txt)
    m ← length(pat)
    n ← length(txt)

    countPat[256] ← {0}
    countWin[256] ← {0}

    for i ← 0 to m-1 do
        countPat[pat[i]]++
        countWin[txt[i]]++
    end for

    for i ← m to n do
        if countPat = countWin then
            print "Pattern found at index", i - m
        end if

        if i < n then
            countWin[txt[i]]++
            countWin[txt[i - m]]--
        end if
    end for
end procedure
```

—

Explanation

- The first window of size m is initialized.
- The window slides one character at a time.
- One character is added and one is removed from the frequency array.
- If both frequency arrays match, the current window is a permutation of the pattern.

—

Correctness

- Since all characters in the pattern are distinct, equal frequency arrays imply identical character sets.
- Therefore, every match corresponds to an occurrence of the pattern or its anagram.

—

Time Complexity Proof

- Initial frequency computation takes $O(m)$ time.
- The sliding window moves across the text once, taking $O(n)$ time.
- Each window update involves constant-time operations.

Since $m < n$, the total time complexity is:

$$O(m+n) = \boxed{O(n)}$$

—

Space Complexity

$$O(1)$$

(Fixed-size frequency array of size 256)

—

Conclusion

The algorithm efficiently finds all occurrences of the pattern and its permutations in linear time using the sliding window technique.

Part a) O(n log n) Disk Intersection Algorithm

Problem Understanding

We have n disks, each defined by center point (x, y) and radius r. Two disks intersect if they share at least one point. We need an $O(n \log n)$ time algorithm to determine if ANY two disks intersect.

Key Insight

Two disks intersect if and only if the distance between their centers is less than or equal to the sum of their radii.

For disk i with center (x_i, y_i) and radius r_i, and disk j with center (x_j, y_j) and radius r_j:

Intersection condition:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq r_i + r_j$$

Or equivalently (avoiding square root):

$$(x_i - x_j)^2 + (y_i - y_j)^2 \leq (r_i + r_j)^2$$

Naive Approach

A brute force approach checking all pairs of disks takes $O(n^2)$ time, which is too slow.

O(n log n) Solution: Sweep Line Algorithm

Algorithm Steps

1. **Create Events** $O(n)$: For each disk i with center (x_i, y_i) and radius r_i:
   - Create a "start" event at x-coordinate: $x_i - r_i$
   - Create an "end" event at x-coordinate: $x_i + r_i$
2. **Sort Events** $O(n \log n)$: Sort all $2n$ events by x-coordinate
3. **Sweep and Check** $O(n \log n)$:
   - Maintain a set of "active" disks (disks whose x-range includes current sweep line position)
   - Use a balanced BST (e.g., interval tree) indexed by y-coordinate intervals
   - For each "start" event:
     - Check if the new disk intersects with any active disk
     - Add the disk to the active set
   - For each "end" event:
     - Remove the disk from the active set

Detailed Algorithm

Algorithm 1 CheckDiskIntersection

Require: Array of disks $D[1..n]$, where each disk has center (x, y) and radius r
Ensure: `TRUE` if any two disks intersect, `FALSE` otherwise

events $\leftarrow \emptyset$
{Step 1: Create events}
for $i = 1$ to n **do**
  Add event $(x_i - r_i, \text{START}, i)$ to events
  Add event $(x_i + r_i, \text{END}, i)$ to events
end for
{Step 2: Sort events}
Sort events by x-coordinate
{Step 3: Sweep line}
active $\leftarrow \emptyset$ {Balanced BST indexed by y-intervals}
for each event e in events **do**
  if e.type = START **then**
    $i \leftarrow e$.disk_id
    $y_{\min} \leftarrow y_i - r_i - r_{\max}$
    $y_{\max} \leftarrow y_i + r_i + r_{\max}$
    candidates $\leftarrow$ active.queryRange($y_{\min}$, $y_{\max}$)
    for each disk j in candidates **do**
      $d^2 \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$
      if $d^2 \leq (r_i + r_j)^2$ **then**
        return `TRUE` {Intersection found}
      end if
    end for
    active.insert(disk i, interval $[y_i - r_i, y_i + r_i]$)
  else{END event}
    active.remove(disk e.disk_id)
  end if
end for
return `FALSE` {No intersection found} =0

Time Complexity Analysis

- Creating events: $O(n)$
- Sorting events: $O(n \log n)$
- Processing events:
  - n disk insertions into BST: $O(n \log n)$ total
  - n disk deletions from BST: $O(n \log n)$ total
  - Range queries and intersection checks: $O(n \log n)$ expected

Total Time Complexity: $O(n \log n)$
Space Complexity: $O(n)$

Why This Works

The sweep line algorithm efficiently prunes the search space by:

- Only checking disks that overlap in the x-dimension (via the sweep line)
- Using spatial indexing (interval tree) for the y-dimension to quickly find candidate disks
- On average, each disk only needs to check $O(1)$ candidates with good spatial distribution