##2017##

1.Solve the following recurrence equation by Master Theorem.T(n)=3T(n/2)+n^2.

Let's solve the recurrence (T(n) = 3T(n/2) + n^2) using the **Master Theorem**.

Recurrence: $T(n) = 3T(n/2) + n^2$

Master Theorem Form: $T(n) = aT(n/b) + f(n)$;Here, $a = 3$, $b = 2$, $f(n) = n^2$

Compute n^(log_b a): $n^{(\log_2 3)} \approx n^{1.585}$

Compare f(n) = n² and n^(log_b a): Since n^2 grows faster than $n^{1.585}$, we are in **Case 3** of the Master Theorem.

Check Regularity Condition: $a\,f(n/b) = 3 \cdot (n/2)^2 = (3/4)n^2 \le c \cdot n^2$ for $c = 3/4 < 1$.Thus, the regularity condition holds.

Conclusion: By Case 3 of the Master Theorem, $T(n) = \Theta(f(n)) = \Theta(n^2)$.

Final Answer: $\Theta(n^2)$

2. What is meant by decision problem and give example?

A decision problem is a computational problem that can be posed as a yes–no question. Specifically, it requires a "yes" or "no" answer (or equivalently, "true" or "false") based on the input. Decision problems are fundamental in complexity theory, as they are used to define complexity classes like P, NP, and NP-complete.

Characteristics:

Input: An instance of the problem (e.g., a graph, a number, a set of constraints).

Output: Either "yes" or "no" (or 1/0).

Well-defined: For every input, there is a correct yes/no answer.

Example 1: Boolean Satisfiability Problem (SAT)

Input: A Boolean formula (e.g., $(x1 \vee \neg x2) \wedge (x2 \vee x3)$).

Question: Is there an assignment of true/false to the variables that makes the entire formula true?

Output: "Yes" if such an assignment exists, "no" otherwise.

SAT was the first problem proven to be NP-complete.

The Master Theorem provides a systematic way to solve recurrence relations of the form:

$T(n)=aT(nb)+f(n)$

3. State and Explain Masters theorem.

Master Theorem: The Master Theorem is used to determine the **time complexity** of divide-and-conquer algorithms of the form:

$T(n)=aT(n/b)+f(n)$

where:

- $a \geq 1$: number of subproblems
- $b > 1$: factor by which the subproblem size reduces
- $f(n)$: extra work outside recursion (divide + combine)

Cases of the Master Theorem

1. **Case 1:**

2. **Case 2:**

3. **Case 3:**

Example:
For Merge Sort:
$T(n)=2T(n2)+O(n)$

Important Notes
The Master Theorem does not apply to all recurrences. For example, T(n)=2T(n/2)+nlogn falls into a gap between cases.
The regularity condition in Case 3 is necessary to ensure that the root term dominates consistently
3.Write a randomized algorithm for computing the value of π also explain the steps.

Randomized Algorithm for Estimating π (Monte Carlo Method)

Algorithm:

1. **Initialize:** Set `count_inside = 0` and total number of points `n`.
2. **Generate Points:** For `i = 1` to `n`:
   a. Generate random point `(x, y)` where `x` and `y` are uniformly distributed in `[0, 1]`.
3. **Check Inside Circle:** If `x² + y² ≤ 1`, increment `count_inside`.
4. **Estimate π:**
   a. $\Pi \approx 4 \times (count\_inside)n$
5. **Return** the estimated value of π.

Explanation of Steps:

1. **Concept:**
   a. Consider a unit square (side=1) and a quarter circle of radius 1 inscribed in it.
   b. Area of square = 1, area of quarter circle = $\pi 44\pi$.
   c. The probability that a random point falls inside the quarter circle is $\pi 44\pi$.
   d. So, $\frac{number\ of\ points\ inside}{total\ points} \approx \pi 4$total pointsnumber of points inside $\approx 4\pi$.
2. **Generate Random Points:**
   a. Use a random number generator to get `x` and `y` in `[0, 1]`.
3. **Check Condition:**
   a. For each point, check if it lies inside the circle (i.e., `x² + y² ≤ 1`).
4. **Estimate π:**

a. Multiply the ratio by 4 to get π.

5. Accuracy:

a. As n increases, the estimate converges to π (by Law of Large Numbers).
b. Error decreases as $O(1/n)$.

Why Randomized?

Uses randomness to approximate a deterministic value.Simple and parallelizable.Classic example of Monte Carlo simulation.

##2018##

1.Compute the prefix function for the pattern ababbab when the alphabet set is (a, b).

The prefix function π[i] = length of the **longest proper prefix** of the substring pattern[0..i] which is also a suffix of this substring.

- "Proper prefix" means not equal to the whole string.

Indexing the pattern (0-based):

i : 0 1 2 3 4 5 6
P : a b a b b a b

Computation:

- **i = 0** → substring "a" → no proper prefix = suffix → π[0] = 0
- **i = 1** → substring "ab" → prefixes: {a}, suffixes: {b} → no match → π[1] = 0
- **i = 2** → substring "aba" → prefixes: {a, ab}, suffixes: {a, ba} → longest match = "a" → π[2] = 1
- **i = 3** → substring "abab" → prefixes: {a, ab, aba}, suffixes: {b, ab, bab} → longest match = "ab" → π[3] = 2
- **i = 4** → substring "ababb" → prefixes: {a, ab, aba, abab}, suffixes: {b, bb, abb, babb} → no match → π[4] = 0
- **i = 5** → substring "ababba" → prefixes: {a, ab, aba, abab, ababb}, suffixes: {a, ba, bba, abba, babba} → longest match = "a" → π[5] = 1
- **i = 6** → substring "ababbab" → prefixes: {a, ab, aba, abab, ababb, ababba}, suffixes: {b, ab, bab, bbab, abbab, babbab} → longest match = "ab" → π[6] = 2

Final Prefix Function:

π=[0,0,1,2,0,1,2]

2. What is the time complexity of Huffman coding algorithm? Justify.

The Huffman coding algorithm has a time complexity of O(n log n), where n is the number of unique characters in the input.

Steps of the Huffman Algorithm:

Frequency Calculation: Count the frequency of each character – O(n).

Build a Min-Heap: Insert all characters with their frequencies into a min-heap (priority queue) – O(n) time.

Construct the Huffman Tree:

Repeatedly extract the two nodes with the smallest frequencies (each extraction is O(log n)).

Merge them into a new node and insert it back into the heap (insertion is O(log n)).

This step is repeated n−1 times (since we start with n nodes and end with one root node).

So, total time for this step is O(n log n).

Generate Codes: Traverse the Huffman tree to assign codes to each character – O(n) (each path is traversed once).

Justification for O(n log n):

The dominant step is the tree construction, which requires n−1 merges.

Each merge involves two extractions (O(2 log n)) and one insertion (O(log n)), so per merge: O(3 log n) = O(log n).

Total for all merges: $(n-1) \times O(\log n) = O(n \log n)$.

Other steps (frequency counting, heap building, code generation) are linear or less.

Thus, the overall time complexity is O(n log n).

3. Given an array z containing a unique real numbers, design and analyze an efficient algorithm that finds out a number in z that is neither minimum nor maximum.

Key Insight

Since all numbers are unique, there are at least three distinct numbers if $n \geq 3$. For $n < 3$, it is impossible to find such an element (so we return an error or None).

We can avoid sorting the entire array (which would take O(nlogn) by using a more efficient approach.

Efficient Algorithm

If $n < 3$, return an error (no solution exists).

Otherwise, compare the first three elements to find one that is neither the min nor the max among them.

This element is guaranteed to be neither the overall min nor the overall max of the entire array.

Why?

The overall min and max must be either smaller or larger than all other elements.

If we consider the first three elements, the one that is neither the smallest nor the largest among these three cannot be the overall min (because there is a smaller one in these three) and cannot be the overall max (because there is a larger one in these three).

Thus, it is safe to return that element.

Steps in Detail:

Let the first three elements be a=z[0],b=z[1], c=z[2].

Compare a, b, and c.

There are 6 possible orderings. For each case, we can pick the middle value.

Example:

If a<b<c, then b is neither min nor max.

If a<c<b, then c is neither min nor max.
Similarly for other cases.

Time Complexity Analysis:

The algorithm only compares the first three elements.
It runs in constant time O(1).
No extra space is used: O(1) space.
Example:

Let z=[5,2,8,1,9].
First three: 5,2,8.
The middle value is 5 (since 2<5<8).
Indeed, 5 is neither the min (1) nor the max (9) of the entire array.

##2019##

1. Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

Randomized algorithms use random choices to achieve good performance on average.
Worst-case analysis may be pessimistic and rare.
Expected running time gives a realistic measure over all possible random choices.
For many randomized algorithms (e.g., randomized quicksort), the worst-case is bad (e.g., O(n^2), but the expected case is efficient (e.g., O(nlog□n).
This reflects typical behavior and is more useful for practice.

2.Can quicksort be made to run in O(n log n) time in the worst case, assuming that all elements are distinct?

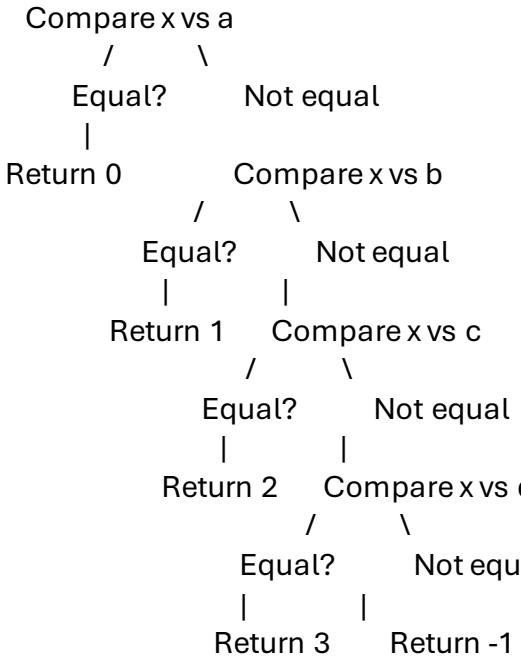
Yes, by using a deterministic pivot selection that guarantees balanced partitions.
Example: Median-of-medians algorithm to choose the pivot in O(n) time.
This ensures the partitions are always balanced, leading to recurrence:

T(n)=2T(n/2)+O(n)=O(nlog□n).

However, the constant factors are large, making it less practical than randomized quicksort.

3. Draw the decision tree for linear search on four elements.

Let the array be A=[a,b,c,d], and we search for key x.
Decision tree steps:
Compare x with a.
If equal, return index 0.
Else, move to next.
Compare x with b.
If equal, return index 1.
Else, move to next.
Compare x with c.
If equal, return index 2.
Else, move to next.
Compare x with d.
If equal, return index 3.
Else, return "not found".
Decision tree structure:



Properties:
Worst-case comparisons: 4 (if not found or at end).
Best-case: 1 (if found at first).
The tree has a linear structure with no branching beyond the next element.

4.Time complexity of KMP algo.

The Knuth-Morris-Pratt (KMP) algorithm is used to find occurrences of a pattern P(length m) in a text T (length n). Its time complexity is:
Preprocessing (building the prefix function for the pattern): O(m)
Matching (searching the pattern in the text): O(n)
Overall time complexity: O(m+n)
Why?
Preprocessing (Prefix Function Calculation):
The prefix function (also known as the failure function) is computed for the pattern.
This is done in O(m) time by iterating through the pattern and using previously computed values.
Matching Phase:
The algorithm traverses the text once, and each character is compared at most twice (due to the use of the prefix function to skip unnecessary comparisons).
Thus, the matching phase runs in O(n) time.
Example:
For pattern P="abc" and text T="ababcabc":
Preprocessing: Compute prefix function for P O(3).
Matching: Scan T once O(8), so total O(11).

<>Naive String Matching Algorithm

Time Complexity: O(m×n)
where m = length of pattern, n = length of text.

Explanation:

For each starting position in the text (there are n−m+1 such positions), the algorithm checks if the pattern matches by comparing all m characters. In the worst case (e.g., when the pattern is "aaa" and the text is "aaaaaaaa"), every character is compared, leading to O(m×n) comparisons.

Rabin-Karp Algorithm

Preprocessing Time: O(m)

(to compute the hash value for the pattern and the first window of the text).

Matching Time:

Worst-case: O(m×n)

(if all windows hash to the same value and we get spurious hits for every position, e.g., with a worst-case hash function).

Average-case: O(m+n)

(with a good hash function that minimizes collisions).

Explanation:

The algorithm uses a rolling hash to quickly compute the hash of each sliding window in constant time. It compares the pattern only when the hash matches. In practice, with a good hash function, the number of spurious hits is low, so the average case is efficient.

##2020##

1. How is string-matching automaton computation different from the computation of prefix function in KMP algorithm?

Both are used in exact string matching, but they differ in approach and computation:

a. String-Matching Automaton (Finite Automaton Approach)

A finite automaton (DFA) is built from the pattern P (length m) to process the text T in one pass.

Computation:

Precomputes a transition function δ for every state (0 to m) and every character in the alphabet.

State q represents the length of the longest prefix of P that is a suffix of the text read so far.

The automaton has m+1 states (0 to m), and state m is the accepting state.

Preprocessing Time: O(m|Σ|) where |Σ| is the alphabet size.

Matching Time: O(n) (each text character is processed in constant time).

Space: O(m|Σ|) (large for big alphabets).

b. Prefix Function in KMP

The KMP algorithm uses a prefix function (also called failure function) π to avoid backtracking in the text.

Computation:

The prefix function π[i] for a pattern P is the length of the longest proper prefix of P[0:i] that is also a suffix.

It is computed in linear time without explicitly building a full automaton.

Preprocessing Time: O(m) (independent of alphabet size).

Matching Time: O(n).
Space: O(m) (only stores the π array).
Example
For pattern P="abab":
Automaton: Builds a DFA with 5 states and transitions for each character.
KMP: Computes π=[0,0,1,2]and uses it to skip comparisons.

2. In randomized closest pair algorithm how is the notion of randomization used?

The randomized closest pair algorithm (based on Rabin's method or using randomized divide-and-conquer) uses randomization to achieve an expected linear-time solution for the closest pair problem in the plane. Here's how randomization is applied:

a. Randomization in the Algorithm

The key idea is to use randomization to avoid worst-case scenarios and ensure good expected performance.

Steps:

Randomly Permute the Points:

First, randomly shuffle the list of points. This ensures that the input order is random, which helps in avoiding worst-case inputs (e.g., already sorted points that could lead to unbalanced partitions).

Recursive Divide-and-Conquer with Randomization:

The algorithm recursively divides the plane into strips and combines results.

By randomizing the order, the splitting lines (e.g., vertical lines) are chosen in a way that the subproblems are expected to be balanced.

Randomized Incremental Construction:

In some versions (e.g., Rabin's algorithm), points are processed in random order.

The closest pair is maintained incrementally. For each new point, we check against a constant number of nearby points (due to randomization, the expected number of checks is small).

b. Why Randomization?

Avoids Worst-Case Inputs: Without randomization, an adversary could provide a pathological input (e.g., points sorted by x-coordinate) that causes the algorithm to degrade to O(n^2).

Expected Linear Time: With random ordering, the expected number of comparisons per point is constant, leading to an overall expected time of O(n).

Simplifies Implementation: Randomization often leads to simpler code compared to deterministic linear-time methods (e.g., using a grid or deterministic divide-and-conquer).

3. What is the significance of computing the expected time T (n) on the average for a randomized algorithm?

Randomized algorithms use random choices during execution. Unlike deterministic algorithms, their performance can vary even for the same input. Analyzing the expected time (average-case over random choices) is crucial for several reasons:

a. Captures Typical Performance

Worst-case might be rare or artificially constructed (e.g., by an adversary).

Expected time reflects the average behavior over all possible random choices, which is often more realistic for practical use.

b. Avoids Pessimism

Worst-case analysis might be overly pessimistic (e.g., quicksort with worst-case O(n^2) is avoidable with randomization).

Expected time (e.g., O(nlogn) for randomized quicksort) shows the efficiency achievable in practice.

c. Guides Algorithm Design

Helps compare randomized algorithms with deterministic ones.

Encourages the use of randomization to simplify algorithms (e.g., randomized primality testing) while maintaining good average performance.

Example: Randomized Quicksort

Worst-case: O(n^2) (if pivot is always smallest/largest).

Expected time: O(nlogn) (with random pivot).

In practice, randomized quicksort is one of the fastest sorting algorithms.

4. If the partition procedure in quicksort divides the array into n/3 and 2n/3 during each recursion of quicksort,express it as a recurrence relation and derive the running time.

Recurrence:

$T(n)=T(n/3)+T(2n/3)+\Theta(n)$

(the Θ(n) is the partition work).

Derivation (short):

At each level of the recursion tree the sum of sizes of subproblems equals n, so the work per level is $\Theta(n)$. The largest child shrinks by a factor $2/3$, so the depth L of the tree is the smallest L with $(2/3)^{(L)}*n = 1$, hence $L=\Theta(\log n)$. Summing $\Theta(n)$ work across $\Theta(\log n)$ levels gives total time

$T(n)=\Theta(n\log n)$.

(Alternate quick justification) Use Akra–Bazzi: solve $(1/3)^p+(2/3)^p=1$. $p=1$ satisfies this, and since the nonrecursive term is $\Theta(n1)$, Akra–Bazzi yields

$T(n)=\Theta(n^{p}\log n)=\Theta(n\log n)$.

5. Arrange the following functions in the increasing order of their rates of growth:

(√2)^n , 2^√n , n(^2) log n, n(log n)^2 , (n log n)^2 , n^(log n) , n^(√n) , n^n , (log n)^n .

(khata)

##2022##

1. Suppose you have algorithms with average time complexities n^2 in log n and low much slower does each of these algorithms get when you

(1) double the input size

(b) increase the input size by one?

We have two algorithms:

- Algorithm A: $T_A(n) = n^2$
- Algorithm B: $T_B(n)=\log n$

We check how the runtime changes when:

(a) Double the input size (n→2n):

- For n^2:

$TA(2n)/TA(n)=(2n)^2/n^2=4n^2/n2=4$

→ Runtime becomes **4× slower**.

- For $\log n$:

$TB(2n)/TB(n)=\log(2n)/\log(n)=(\log n+\log 2)/\log n=1+(\log 2/\log n)$

→ Growth is very small. For large n, this ratio tends to **1**. At small n, it's slightly bigger than 1.

(b) Increase input size by 1 (n→n+1):

- For n^2:

$TA(n+1)/TA(n)=(n+1)^2/n^2=n^2+2n+1/n^2=1+(2/n)+(1/n^2)$

→ Runtime increases by about $2/n$ **fraction** (very small when n is large).

- For $\log n$:

$TB(n+1)/TB(n)=\log(n+1)/\log(n)$

For large n:

$\log(n+1)$≈logn+(1/n)

So the ratio ≈ $1/n\log n$, which is **almost no change**.

2. Let A[1….. 95)] be a sorted array of integers. In order to find the 17th smallest integer how many calls to Select algorithm will be required?

To find the kth smallest element in a sorted array, we can use the Selection algorithm (e.g., the median-of-medians algorithm or quick select). However, since the array is already sorted, we can directly access the kth element in O(1) time without any calls to a selection algorithm.

But the question asks for the number of calls to the Select algorithm (which typically refers to a recursive selection method like quick select or the median-of-medians algorithm) if we were to apply it. However, for a sorted array, the selection algorithm is unnecessary.

##2024##

1. Find the number of possible parenthesization of matrix chain multiplication for the sequence of matrices <M1, M2, M3, M4 >

For 4 matrices the number of full parenthesizations is the Catalan number $C_{3}=5$.

The 5 ways (with matrices $M1,M2,M3,M4$) are:

1. $((M1M2)M3)M4$
2. $(M1(M2M3))M4$
3. $(M1M2)(M3M4)$
4. $M1((M2M3)M4)$
5. $M1(M2(M3M4))$.

2. Derive the time complexity of the divide and conquer strategy-based algorithm to solve the closest pair algorithm.

The **closest pair of points** problem (in 2D) can be solved by a divide-and-conquer approach. Let me break down the complexity derivation:

Step 1: Divide

- Sort points by x-coordinate: $O(n \log n)$.
- Recursively split into two halves of size $n/2$.

Step 2: Conquer

- Recursively find the closest pair in each half:

$T(n) = 2T(n/2) + (\text{merge step})$

Step 3: Combine (Merge Step)

- After getting $\delta = \min(\delta L, \delta R)$, check across the dividing line.
- Only points within distance δ from the line are considered.
- Each point is compared with at most 7 others in its δ-strip (packing argument).
- Sorting by y-coordinate can be maintained in $O(n)$ time at each level.
- So merge step = $O(n)$.

Step 4: Recurrence Relation

$T(n) = 2T(n2) + O(n)$

Step 5: Solve Recurrence

By Master Theorem:

- $a = 2, b = 2 \Longrightarrow n \text{^} \log b\_a = n$
- Since the combine step is $O(n)$, case 2 applies.

$T(n) = O(n \log n)$

2024 Midsem

1. Define the prefix function used in finite automata based string matching.

1. Prefix function (π-function)

- The **prefix function** for a pattern $P[1..m]$ tells us the length of the **longest proper prefix** of $P[1..q]$ that is also a suffix of $P[1..q]$.
- It is used in **KMP (Knuth–Morris–Pratt)** automaton construction to know how far we can "fall back" when a mismatch happens.

Example: Pattern $P = "ababaca"$

We compute π for each prefix:

- π[1] = 0 (no proper prefix = suffix)
- π[2] = 0 ("ab")

- π[3] = 1 ("aba": longest border = "a")
- π[4] = 2 ("abab": border = "ab")
- π[5] = 3 ("ababa": border = "aba")
- π[6] = 0 ("ababac": no border)
- π[7] = 1 ("ababaca": border = "a")

So: **π = [0,0,1,2,3,0,1]**

2. Suffix function (δ-function in finite automaton)

- The **suffix function** δ(q, a) tells us: if we are in state q (matched q characters) and see character a, then δ(q, a) = length of the **longest prefix of P** that is also a suffix of $P[1..q]+a$.
- This is the transition function of the finite automaton for pattern matching.

Example: Same $P="ababaca"$.

Suppose we are in state q=4 (already matched "abab"), and the next input is a:

- The string is "ababa".
- Longest prefix of P that is also suffix of "ababa" = "ababa" (length 5).
- So δ(4, 'a') = 5.

✅ **Summary:**

- **Prefix function (π):** table of fallback values for each position in the pattern.
- **Suffix/transition function (δ):** defines the automaton transitions based on next input character, using prefix-suffix relationships.

3. Differentiate between Randomized and Deterministic algorithm.

Randomized Algorithm

- Uses random numbers during execution.
- Output or running time may vary across runs.
- Guarantees are usually **probabilistic** (e.g., "correct with high probability" or "expected time").

- Example: **Randomized QuickSort** (chooses pivot randomly).

Deterministic Algorithm

- No randomness; behavior is fixed for given input.
- Always produces the same output in the same running time.
- Guarantees are **absolute**, not probabilistic.
- Example: **MergeSort** (always splits array in half).

In short: **Randomized = uncertainty in path, but often faster in practice; Deterministic = fixed path, predictable behavior.**

4. Does Randomization improves efficiency for QuickSort and Selection Algorithm? Justify your answer.

Yes

Quicksort

- **Without randomization (deterministic pivot):** Worst-case partitioning (e.g., already sorted array with first element as pivot) gives

  $T(n)=T(n-1)+\Theta(n)=\Theta(n^2)$

- **With randomization (random pivot):** Each pivot is equally likely → avoids consistently bad splits. The expected recurrence becomes

  $T(n)=T(k)+T(n-k-1)+\Theta(n)$

which solves to **expected** $\Theta(n\log n)$.

👉 **Randomization reduces worst-case dependence on input ordering.**

Selection Algorithm (QuickSelect)

- **Without randomization (deterministic pivot):** If pivot choice is poor, recursion depth can be n, leading to **worst-case** $\Theta(n^2)$.
- **With randomization (random pivot):** Expected split is balanced enough → **expected** $\Theta(n)$time for finding the k-th smallest element.

Justification: Randomization prevents adversarial inputs from forcing bad pivots, ensuring **good expected performance** for both QuickSort and QuickSelect.

5. Give the Dynamic Programming formulation for the all pair shortest path problem.

PB Ma,am PYQ 2(chatgpt)

We define:

Recurrence:

Final answer:

6. If n element input to the Select algorithm (finding kth smallest element) is already sorted and I want to find the n/4 smallest element, after how many recursions will you get the n/4 th smallest element?

Since the array is **already sorted**, the 4-th smallest element is just at index .

👉 No recursion is needed; it can be directly accessed in $O(1)$.

7. What is the total number of paranthesization possible for a sequence of 5 matrices A1,A2, A3, A4,A5.

Total number of parenthesizations of 5 matrices:

For n matrices, the number of ways = **Catalan number** C_{n-1} .

Here, $n=5$:

C4=1/n+1(2nCn)=70/5=14

👉 So, total = **14 parenthesizations**.

- *(((A1A2)A3)A4)A5*
- *((A1A2)A3)(A4A5)*
- *((A1(A2A3))A4)A5*
- *(A1((A2A3)A4))A5*
- *(A1(A2(A3A4)))A5)*
- *(A1A2)((A3A4)A5)*
- *(A1(A2A3))(A4A5)*
- *A1((((A2A3)A4)A5))*
- *A1(((A2A3)(A4A5)))*
- *A1((A2(A3A4))A5)*
- *A1((A2A3)(A4A5))*
- *A1(A2((A3A4)A5))*
- *A1(A2(A3(A4A5)))*
- *(A1(A2(A3A4)))A5*

8.A contiguous subsequence of a list 5 is a subsequence made up of consecutive elements of S. for example if S is 5,15,-30,10,-5,40,10, then 15,-30,10 is contiguous subsequence but 5,15,40 is not. You are given a list of numbers a formulate a dynamic programming formulation to compute a contiguous subsequence whose sum is maximum. For the example. 10, -5, 40, 10 with sum 55 is maximum.

Maximum contiguous subsequence — dynamic programming (Kadane) with reconstruction

Nice — two short answers.

Max contiguous subsequence (DP / Kadane formulation)

Let A[1..n]. Define M[i] = maximum sum of a contiguous subsequence **ending at** index i.

Base: M[1] = A[1]. Recurrence:

$M[i]=\max(A[i], M[i-1]+A[i]) (i\geq 2)$

Answer = max_{1≤i≤n} M[i]. Runs in O(n).

Example A = [10,-5,40,10]: M = [10,5,45,55] → max = **55**.

9. Show the steps of divide and conquer algorithm to multiply two long binary integers 10011011 and 10111010.

Divide-and-conquer multiplication of x = 10011011 and y = 10111010
Use 8-bit split into 4-bit high/low:

- x_H = 1001 (9), x_L = 1011 (11)

- y_H = 1011 (11), y_L = 1010 (10)

Compute the four products (recursively / directly here):

- A*C = x_H*y_H = 9*11 = 99 → 1100011
- A*D = x_H*y_L = 9*10 = 90 → 1011010
- B*C = x_L*y_H = 11*11 = 121 → 1111001
- B*D = x_L*y_L = 11*10 = 110 → 1101110

Combine with shifts (base = 2^4):

x·y=(AC)·2^8+(AD+BC)·2^4+(BD).
Compute AD+BC = 90+121 = 211 (11010011). Now the aligned binary pieces (15 bits):

AC << 8 = 1100011 00000000 = 110001100000000

(AD+BC)<<4= 000110100110000

BD = 000000001101110

---

sum = 111000010011110(decimal 28830)

10. Hash in Rabin-Karp String Matching Algorithm

The **Rabin-Karp algorithm** uses **hashing** to efficiently find occurrences of a pattern (P) (length (m)) in a text (T) (length (n)). The key idea is to compare the hash value of the pattern with the hash value of each substring of the text of length (m). If the hash values match, then a character-by-character comparison is done to confirm the match.

1. Role of Hash Function

- The hash function converts a string of length (m) into an integer value.
- A good hash function minimizes **collisions** (different strings having the same hash) and allows efficient computation of the hash for sliding windows.

2. Rolling Hash: Efficient Hash Computation

Rabin-Karp uses a **rolling hash** to compute the hash for the next text substring in constant time (O(1)) after computing the first hash.

Common Hash Function: Polynomial Rolling Hash

Let the pattern (P = p_0 p_1 … p_{m-1}) and a text substring (S = t_i t_{i+1} … t_{i+m-1}).

- Choose a base (b) (e.g., 256 for ASCII) and a modulus (q) (a large prime to avoid overflow).
- The hash (H(S)) for a string (S) is computed as: [H(S) = (s_0 \cdot b^{m-1} + s_1 \cdot b^{m-2} + … + s_{m-1}) \mod q]

Example:

For (S = "abc"), (b=256), (q=101): [H("abc") = (97 \cdot 256^2 + 98 \cdot 256^1 + 99) \mod 101]

4. Steps of Rabin-Karp

1. **Precompute:**
    a. Compute (H(P)) for the pattern.
    b. Precompute (b^{m-1} \mod q) (for rolling hash).
2. **Compute initial hash** (H_0) for (T[0 … m-1]).
3. **For each index (i) from 0 to (n-m):**
    a. If (H_i == H(P)), compare (T[i … i+m-1]) with (P) character-by-character.
    b. Update the hash to (H_{i+1}) using the rolling hash.

5. Why Use Hashing?

- **Speed:** Comparing hash values is faster than comparing strings.
- **Efficiency:** The rolling hash allows each new hash to be computed in (O(1)) time.
- **Average-case time complexity:** (O(m + n)) (with good hash function).

11. Why do we use randomization algorithm?

Randomized algorithms use random choices during their execution to achieve efficiency, simplicity, or both. They are widely used in various domains for several key reasons:

a. Avoid Worst-Case Inputs

- **Problem:** Deterministic algorithms may have poor worst-case performance due to adversarial inputs.

- **Solution:** Randomization makes the algorithm behavior unpredictable, preventing adversaries from forcing worst-case scenarios.
- **Example:** Randomized Quicksort chooses a random pivot to avoid the worst-case (O(n^2)) time that occurs with sorted input.

b. Simplify Algorithms

- Randomized algorithms are often simpler and more elegant than their deterministic counterparts.
- **Example:** The randomized algorithm for the closest pair of points is simpler than the deterministic (O(n log n)) solution.

c. Improve Average-Case Performance

- Many randomized algorithms have excellent expected performance, even if worst-case performance is poor.
- **Example:** Randomized Quicksort has an expected time of (O(n log n)) with small constants.

12.Why do we compute expected time complexity for randomized algorithm and not It's worst case running time?

Randomized algorithms use random choices during execution, leading to variable performance. Here's why we focus on **expected time complexity** rather than worst-case:

1. Worst-Case is Often Rare and Avoidable

- Randomized algorithms are designed to **avoid worst-case inputs** by using randomness.
- For example, in **Randomized Quicksort**, choosing a random pivot prevents an adversary from forcing the (O(n^2)) worst-case (which happens with a fixed pivot strategy).
- The worst-case might be so improbable that it rarely occurs in practice.

2. Expected Time Reflects Typical Performance

- The **expected time** (average over all random choices) gives a realistic measure of how the algorithm performs **on average**.
- This is more useful for practical applications where worst-case scenarios are unlikely.
- Example: Randomized Quicksort has expected time (O(nlog n)) with very low constants, making it one of the fastest sorting algorithms in practice.

3. Worst-Case Might Be Pessimistic

- For many randomized algorithms, the worst-case is still bad (e.g., (O(n^2)) for Quicksort), but the probability of that worst-case is exponentially small.
- Analyzing worst-case time would be overly pessimistic and not reflective of real-world usage.

13. How to use randomization to find the k-th smallest element from an array linear expected time complexity?

The randomized algorithm for selection (often called quickselect) is based on the same partitioning idea as quicksort but only recurses on one side. It achieves an expected time complexity of O(n).

Algorithm Steps

Choose a random pivot element from the array.
Partition the array around the pivot:
All elements < pivot go to the left.
All elements > pivot go to the right.
The pivot ends up in its correct sorted position (index p).
Recurse on the appropriate subarray:
If p=k, return the pivot (it is the k-th smallest).
If p>k, recurse on the left subarray.
If p<k, recurse on the right subarray and adjust k (since we skip p+1 elements).

Why Expected Time is O(n)?

Each partitioning step takes O(n) time.
The pivot is chosen randomly, so it approximately splits the array in half on average.
The expected number of recursive calls is O(log n), but the total work decreases geometrically.
The overall expected time is:$T(n)=T(n/2)+O(n) \implies T(n)=O(n)T$
(This recurrence solves to O(n) because the problem size is halved on average.)

14. Let 'g' denote the size of group in divide and conquer select algorithm that find k-th Smallest element. Derive the running time complexity in term of g. What happen if g is too large compared to 5.

Divide and Conquer Select Algorithm (Median-of-Medians)

The algorithm finds the (k)-th smallest element in (O(n)) worst-case time. It groups the array into subsets of size (g), finds the median of each group, recursively finds the median of these medians, and uses it as a pivot.

1. Recurrence Relation in Terms of (g)

Let (T(n)) be the time complexity.

Steps:

1. Divide the (n) elements into [ceil n/g] groups of size (g) (except possibly the last group).
2. Find the median of each group (each takes (O(g)) time, so total (O(n))).
3. Recursively find the median (M) of the [ceil n/g] medians: (T(n/g)).
4. Partition the array around (M): (O(n)).
5. Recurse on at most (3n/4) elements (since (M) is a good pivot that eliminates at least (n/4) elements).

2. Solving the Recurrence

We want (T(n) = O(n)). This requires that the recursive calls shrink sufficiently fast.

3. What If (g) Is Too Large?

- If (g) is too large (e.g., (g = 10)), the number of groups (\lceil n/g \rceil) is smaller, so the recursive call (T(n/g)) is cheaper.
- However, the constant factors may increase because finding the median of a large group (size (g)) takes more time ((O(g \log g)) if sorted, but typically we use a constant-time method for small (g)).
- The recurrence still holds as long as (g > 4), so (T(n) = O(n)).

But if (g) is very large (e.g., (g = n)), then:

- There is only one group, so the median of the group is the median of the entire array.
- Finding the median of a large group naively would take (O(n log n)) time (if sorted), which is expensive.
- The recurrence becomes: [T(n) = T(1) + T(3n/4) + O(nlog n)] This solves to (T(n) = O(nlog n)), which is worse.

4. Why (g = 5) Is Optimal?

- (g = 5) is the smallest odd number greater than 4 that ensures linear time.
- It balances the cost of computing group medians and the size of the recursive call.
- Larger (g) (e.g., 7, 9) also work, but the constant factors increase.

5. Conclusion

For (g >= 5) and odd, (T(n) = O(n)).

- If (g) is too large (e.g., (g = n)), the algorithm becomes (O(nlog n)).
- **Optimal choice:** (g = 5) or (g = 7) (to avoid worst-case overhead).

15. Design an efficient randomized algorithm that generates a random permutation of the integers 1,2,….n. Assume that you have access to a fair-coin. Analyze the time complexity of your algorithm.

We can use the **Fisher–Yates shuffle (Knuth shuffle)**, with randomness generated from coin flips.

Algorithm (Randomized Shuffle)

1. Start with the array $A=[1,2,\ldots,n]$.
2. For $i=n$ down to 2:
    a. Generate a random integer $j \in [1,i]$ using fair-coin flips.
    b. Swap $A[i]$ and $A[j]$.
3. Output A.

Correctness

- Each element is equally likely to be placed in any of the n positions.
- By induction, the algorithm generates all $n!$ permutations with equal probability.

Using Fair Coin

- To generate a uniform random integer $j \in [1,i] j \backslash in\ [1,i]$, repeatedly flip coins to generate binary digits until you obtain a number in the range $[1,i][1,i]$.
- This ensures unbiased selection. Expected $O(\log i)$ coin flips per random number.

Time Complexity

- **Swapping loop:** $O(n)O(n)$.
- **Random index generation:** Each step costs expected $O(\log i)$, so total expected complexity

$O(\sum i=1 \text{ to } n * \log i) = O(n \log n)$.
If we assume direct access to uniform random numbers in $[1,i]$, the complexity improves to $\boldsymbol{O(n)}$

16. Given an unsorted array of 50 elements, design an algorithm with O(n) time complexity to print the least 25 elements.

We need the 25 smallest elements from 50 numbers in O(n).

Idea

- Use the **Quickselect algorithm** (selection algorithm based on partitioning, same as in Quicksort).
- Quickselect can find the k-th smallest element in expected $O(n)$.
- Then, partitioning gives us all elements smaller than or equal to that pivot.

Algorithm (Quickselect for 25 smallest)

1. Input array $A[1..50]$.
2. Use **Quickselect(A, k=25)** to find the 25-th smallest element x.
    a. Partition array around pivot until pivot is the 25-th element.
3. Traverse A, print all elements $\leq x$.
4. (If duplicates around x cause more than 25 outputs, print exactly 25 smallest by selecting from that subset.)

Complexity

- Quickselect runs in expected $O(n)$.
- One additional linear scan prints results.
- Total: $O(n)$.

✅ **Answer:**

Use Quickselect to find the 25-th smallest element in the array, then print all elements $\leq$ it. This algorithm runs in expected **linear time** $O(n)$.

NETWORK FLOW

1.Define Flow in Flow-networks.

Flow Networks:
A flow network is a directed graph G = (V, E) in which each edge (u, v) ∈ E has a nonnegativecapacity c(u, v) ≥ 0. There are two special vertices: a source s, and a sink t.

Flows:
Given an s-t network, a flow (also called an s-t flow) is a function f that maps each edge to a nonnegative real number and satisfies the following properties:

Capacity Constraint: For all (u, v) ∈ E, f(u, v) ≤ c(u, v).
Flow conservation (or flow balance): For all v ∈ V - {s, t}, the sum of flow along edges into v equals the sum of flows along edges out of v.

f in(v) = f out(v), for all v ∈ V - {s, t}.

3.Given a flow network $G=(V,E)$ where $V=\{s,u,t,v\}$ and $E=\{(s,u,2),(u,t,4),(s,v,4),(v,t,2)\}$ Find the **maximum flow** in the network G, showing all steps.

Deepseek

1. State and prove the *Max-flow Min-cut Theorem*.
2. **Give the time complexity of the *Ford-Fulkerson algorithm* and justify it.**

The **Ford–Fulkerson algorithm** finds the maximum flow in a flow network by repeatedly finding augmenting paths in the residual graph and pushing flow along them until no augmenting path exists.

1. Time Complexity Statement

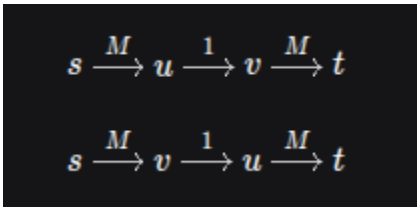
The time complexity is: [O(|E|.f_max)]
where:(|E|) = number of edges,(f_max) = value of the maximum flow

2. Justification

- **Each augmentation** increases the flow by at least 1 unit (if capacities are integers).
- So, there are at most (f_max) augmentations.
- **Finding an augmenting path** using DFS or BFS takes (O(|E|)) time in a graph with (|V|) vertices and (|E|) edges (since BFS/DFS is (O(|V| + |E|)) and typically (|E|>=|V|) in flow networks).
- Hence total time = (O(|E|.f_max)).

3. Example to Illustrate Dependency on (f_max)

Consider a graph:



If (M) is large, Ford–Fulkerson might repeatedly alternate flow between the two central edges (u->v) and (v->u), taking (O(M)) iterations, even though (|E|) is small.
Here (f_max = 2M), so time is (O(M.|E|)), which is large if (M) is large.

3. Provide an example of a **flow network** where the *Ford-Fulkerson algorithm* takes the **maximum number of iterations**.

To see this, consider the example shown in Fig. 56. If the algorithm were smart enough to send flow along the topmost and bottommost paths, each of capacity 100, the algorithm would terminate in just two augmenting steps to a total flow of value 200. However, suppose instead that it foolishly augments first through the path going through the center edge. Then it would be limited to a bottleneck capacity of 1 unit. In the second augmentation, it could now route through the complementary path, this time undoing the flow on the center edge, and again with bottleneck capacity 1. Proceeding in this way, it will take 200 augmentations until we terminate with the final maximum flow.

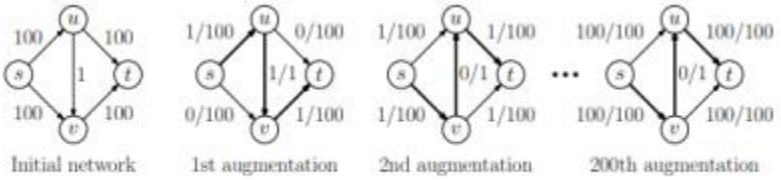


Fig. 56: Bad example for Ford-Fulkerson.

4. Let f and f' be two feasible (s,t)-flows in a graph G, with $|f'|>|f|$.
Prove that there exists a feasible (s,t)-flow of value |f'|−|f| in the **residual graph** Gf.

Flow Difference:

Let Δf = f '− f be the flow difference between f' and f. Clearly, Δf is also a feasible flow in the network because both f and f ' are feasible, and subtracting one feasible flow from another yields another feasible flow.

Residual Network:

The residual network Gf for a flow f is constructed by considering the residual capacities for each edge. The residual capacity on an edge (u , v) in the residual network is defined as:

Cf (u , v) = c(u , v) − f(u , v)

where c(u , v) is the original capacity of the edge (u , v) and f(u ,v) is the flow through that edge .

In the residual network Gf, there are two types of edges:

Forward edges: If there is a flow f(u , v) the residual capacity is c(u , v) − f u , v).

Backward edges: If there is a flow f(u , v), the residual capacity is f(u , v) on the reverse edge (v , u).

Feasibility of Δ f : To show that Δf is a feasible flow in the residual network Gf, we need to check the flow conservation and capacity constraints:

Capacity constraint: For each edge (u , v), the flow Δf(u , v) must respect the residual capacity,
meaning:

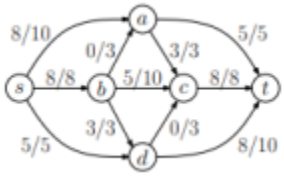
0≤Δf(u , v)≤Cf(u , v)

Since f ' and f are both feasible flows, it follows that the flow difference Δf = f ' - f does not violate the capacity constraints.

Flow conservation: For each vertex v, the flow conservation condition must hold for Δ f. This means that the total incoming flow to v should equal the total outgoing flow from v. Since f and f '
are both feasible flows, their difference Δ f will also satisfy flow conservation at each vertex

5. *True or False:* Let G be a flow network with source s, sink t, and integer capacities. If f is a maximum $s-t$ flow, then every edge out of s must be saturated with flow.
   In a flow network, having a maximum flow ff does not necessarily mean that every edge out of the source ss is saturated (i.e., has flow equal to its capacity). The maximum flow ff is defined as the largest amount of flow that can be sent from the source ss to the sink t, subject to the capacity constraints on the edges.

   Saturation of Edges: An edge is said to be saturated if the flow through that edge equals its capacity. While it is true that some edges leaving the source may be saturated in a maximum flow scenario, it is not required for all edges out of ss to be saturated. The flow can be distributed among multiple edges, and some edges may carry less than their full capacity.



6. Given $G(V,E)=\{s,u,v,t\}$ and edge set $E=\{(s,u,1),(s,v,1),(u,t,1),(v,t,1),(u,v,1)\}$ list all **minimum s–t cuts** and prove that **Min-cut = Max-flow**.
   Deepseek
7. Provide an example in which the Ford–Fulkerson algorithm makes O(f) iterations, where f is the maximum flow.

We want an example where Ford–Fulkerson, using a *poor choice* of augmenting paths, requires **Θ(f)** iterations, where (f) is the value of the maximum flow.

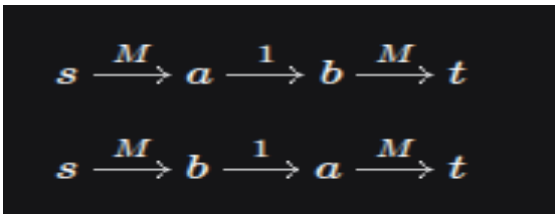
1. Understanding the requirement

Ford–Fulkerson's worst-case time is (O(|E|.f_max).
This bound is tight if each augmentation increases the flow by only 1 unit, and there are (f_max) augmentations.
So we need a graph and a sequence of augmenting paths such that:

- Each path pushes only 1 unit of flow.
- The number of iterations = (f_max).

2. Classic "bad" example

Consider this graph: Vertices: ({s, u, v, t})
Edges: [(s, u, 1), (u, v, 1),(v, t, 1),(s, v, 1),(u, t, 1)]
But that's not the worst case — let's use the standard example from textbooks:
Graph:



Where (M) is a large integer.

Vertices: (s, a, b, t)
Edges with capacities:

1. ((s, a, M))
2. ((s, b, M))
3. ((a, b, 1))
4. ((b, a, 1))
5. ((a, t, M))
6. ((b, t, M))

3. Poor path selection

Max flow = (2M) (can send (M) along (s->a-t) and (M) along (s->b->t)).
But if we choose augmenting paths that alternate flow through the middle edges ((a, b)) and ((b, a)), each path pushes only 1 unit.
Step-by-step bad iteration sequence:
Start: flow = 0.

1. Augmenting path: (s->a->b->t)

Push 1 unit.
Residual: ((a, b)) cap 0, reverse ((b, a)) cap 1.

2. Augmenting path: (s->b->a->t)

Push 1 unit.
Residual: ((b, a)) cap 0, reverse ((a, b)) cap 1.

3. Augmenting path: (s->a->b->t)

Push 1 unit.
Residual: ((a, b)) cap 0, reverse ((b, a)) cap 1.

4. Augmenting path: (s->b->a->t)

Push 1 unit.

... and so on.

Each odd iteration uses (a ->b), each even iteration uses (b->a), each pushing 1 unit.

After 2 iterations, total flow = 2.

We need (2M) iterations to reach max flow (2M).

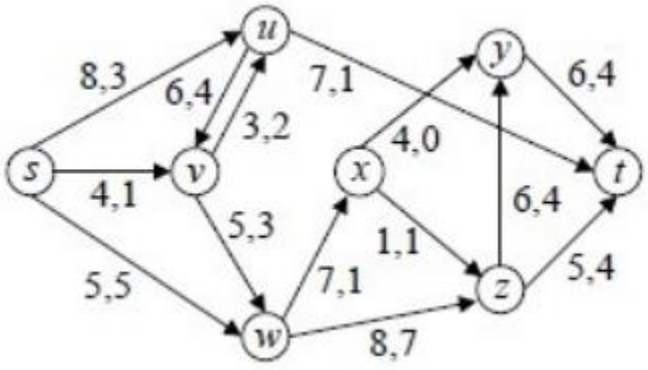
Thus **number of iterations = (2M = f_max)**.

4. Conclusion

This matches (O(f)) iterations exactly.

[See graph with edges: (s,a,M), (s,b,M), (a,b,1), (b,a,1), (a,t,M), (b,t,M)]With alternating paths (s ->a->b-t) and (s->b->a->t), Ford–Fulkerson takes (2M) iterations for max flow (2M).

8. Consider the given network flow (with source s and sink t). The capacity c(e) and current flow amount f(e) are shown for each edge e.
   Run the Ford–Fulkerson algorithm until the maximum flow is computed.



9. Give an example of a small network where the Ford–Fulkerson algorithm can take exactly mC iterations, where m is the number of edges and C is the maximum capacity.

- m = number of edges
- C = maximum capacity of an edge
- Each iteration pushes 1 unit of flow

1. Understanding the bound

Ford–Fulkerson's worst-case number of iterations can be O(m.C) if each augmenting path increases flow by 1 and the max flow is m.C.

But max flow f_max might be less than m.C, so we need f_max= m.C for the worst case.

So we need:

- m edges,Each with capacity C ,Max flow = mC ,Each augmentation sends 1 unit

2. Constructing the example

Let's take a simple graph:
Vertices: (s, t) and m parallel edges from s to t , each with capacity C . Then max flow = mC .
If we choose augmenting paths poorly:Always use one edge at a time, sending 1 unit each time, we need mC iterations.But here each augmenting path uses only one edge — that's valid.
Example: Let m = 2, C = 3.
Edges:
e_1: s->t , capacity 3
e_2: s->t, capacity 3
Max flow = 6.
Bad sequence:
Iteration 1: push 1 along e_1
Iteration 2: push 1 along e_2
Iteration 3: push 1 along e_1
Iteration 4: push 1 along e_2
…
After 6 iterations, flow = 6.
Number of iterations =mC = 2.3 = 6.

3. Generalization

For m edges s ->t , each capacity C,
max flow = mC .
If each augmentation sends 1 unit and cycles through edges one by one,iterations =mC
So: [G = (V={s, t},E = { (s,t,C){ repeated m times})]

With augmentation sequence: repeatedly pick edges in round-robin fashion, sending 1 unit each time.

10. Prove or disprove the statement: "If all capacities in a network are distinct, then there exists a unique flow function that gives the maximum flow."
    Deepseek
11. State the properties that must be satisfied by a flow in a flow network.

In a flow network (G = (V, E)) with source (s), sink (t), and capacity function (c: E->{R_>= *0}*), *a **flow** (f: E->R_>=0}*) must satisfy the following properties:

1. Capacity Constraint

For every edge ((u, v) ∈ E): **[0<=f(u, v)<=c(u, v)]**
The flow on an edge cannot exceed its capacity and must be non-negative.

2. Flow Conservation

For every vertex (u ∈ V \{s, t}):

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$$

That is, the total flow entering (u) equals the total flow leaving (u) (for intermediate nodes).

3. Flow Value

The **value** of the flow (|f|) is defined as:

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

Net flow out of source (s).

Equivalently, net flow into sink (t):

$$|f| = \sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v)$$

12. Given a flow F in a network, prove that the flow out of the source equals the flow into the sink.

    For any flow (F) in a network, the **net flow out of the source** equals the **net flow into the sink**.

1. Definitions

Let (G = (V, E)) be a flow network with source (s), sink (t), and flow (f: E ->R_>=0}).
Net flow out of (s):

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

Net flow into (t):

$$\sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v)$$

2. Flow conservation for all vertices except (s) and (t)

For all (u ∈ V \{s, t}):

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$$

3. Sum over all vertices

Consider the sum over all vertices (u ∈ V) of:

$$\left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u)\right)$$

By flow conservation, for (u not equal to s, t), this difference is 0.

For (u = s): this difference is (|f|) (net flow out of (s)).

For $u = t$:
Outflow from t = $\sum_v f(t,v)$
Inflow to t = $\sum_v f(v,t)$
So difference = $\sum_v f(t,v) - \sum_v f(v,t)$ = **negative** of net flow into t.

Let $\text{net\_into}(t) = \sum_v f(v,t) - \sum_v f(t,v)$.
Then for $u = t$, the difference out $-$ in = $-\text{net\_into}(t)$.

4. Total sum is zero

The sum over all (u ∈ V) of (outflow - inflow) = 0, because each (f(x, y)) appears once as outflow from (x) and once as inflow to (y) with opposite signs.

So:

$$\sum_{u \in V} \left[\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u)\right] = 0$$

But this equals:

$$|f| + (-\text{net\_into}(t)) + \sum_{u \neq s,t} 0 = 0$$

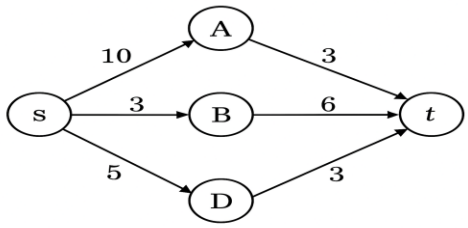
Thus:

$$|f| - \text{net\_into}(t) = 0$$

$$|f| = \text{net\_into}(t)$$

This is a fundamental property of flow networks, arising from flow conservation at all intermediate nodes and the antisymmetry of the flow sum over all vertices.

13. Given a network flow diagram, identify all minimum s–t cuts and verify that the value of the max-flow equals the capacity of the min-cut.



Edges (capacity): s->A:10;A->t:3;s->B:3;B->t:6;s->D:5;D->t:3.

1) Produce a feasible flow and its value (upper bound on max flow).

Send flows along the three simple s-to-t paths:

- s->A->t: send 3 (limited by A->t=3).
- s->B->t: send 3 (limited by s->B=3).
- s->D->t: send 3 (limited by D->t=3).

All other direct paths to t are unavailable. Total flow value = 3+3+3=9.

2) Show this flow is maximum by finding a cut of capacity 9.

Construct residual graph after the flow above:

- Remaining forward capacity: s->A has 10-3=7, s->B has 3-3=0, s->D has 5-3=2.
- A->t and D->t are saturated (0 forward), B->t has 6-3=3 forward. Reverse edges from A,B,D to s carry the 3 units of flow back.

Find vertices reachable from s in the residual graph: starting at s you can go to A via 7 and to D via 2 but not to B (its forward capacity is 0). So R={s,A,D}.

Find vertices that can reach t in the residual graph: B can reach t (forward capacity 3), so W={B,t}.

The canonical cut from the max-flow residual analysis is (R, V\ R) = {s,A,D},{B,t}. Its capacity equals the sum of capacities of all original edges from R to (V\ R):

- s->B 3, A->t 3, D->t 3. No other edges go from R to {B,t}.

So cut capacity =3+3+3=9.

3) Conclude max-flow = min-cut and list all minimum cuts.

By the max-flow / min-cut theorem the flow value 9 equals the capacity of the cut {s,A,D},{B,t}. From the residual reachability argument all minimum s–t cuts are exactly the sets (S) with

$$R \subseteq S \subseteq V \setminus W.$$

Here R={s,A,D} and (V \ W={s,A,D}), so the only set satisfying $R \subseteq S \subseteq V \setminus W$ is S=R. Therefore the cut {s,A,D},{B,t} is the unique minimum s–t cut, and its capacity equals the maximum flow 9.

14. Suppose that flow network G contains an edge (u,v) and we create a flow network G` by creating a new vertex x and replacing (u,v) with new edges (u,x) and (x,v) with capacity c(u,x)=c(x,v)=c(u,v).Does a maximum flow in G` have the same value as the maximum flow in G? Justify your answer.

1. Understanding the transformation

We start with G, containing edge (u, v) with capacity c(u, v).
We create (G') by:

- Adding a new vertex (x)
- Removing (u, v)
- Adding (u, x) with capacity c(u, v)
- Adding (x, v) with capacity c(u, v)

So, the capacity through the "chain" u->x->v is limited by c(u, v) on both edges.

2. Intuition

The transformation doesn't change the **bottleneck capacity** between u and v, because in G', the flow from u->v is still constrained by c(u, v) (both edges have that capacity, so effectively the u-v capacity remains c(u, v).
So any flow in G that uses k <=c(u, v) units on (u, v) can be routed in G' as k units along (u->x ->v), and vice versa.

3. Proof of equivalence

Part 1: Max flow in G ≤ Max flow in G'

Let f be a max flow in G . Define a flow f' in G' as:

- f'(e) = f(e) for all edges e in G except (u, v)
- f'(u, x) = f(u, v)
- f'(x, v) = f(u, v)

Check capacity constraints in G' :

- f'(u, x) = f(u, v) <=c(u, v) = c'(u, x)
- f'(x, v) = f(u, v) <= c(u, v) = c'(x, v)

Flow conservation at x : inflow f'(u, x) = outflow f'(x, v), so satisfied.
Thus f' is valid in G' with same value as f .
So max-flow(G) <=max-flow(G').

Part 2: Max flow in G' ≤ Max flow in G

Let f' be a max flow in G' . Define a flow f in G as:

- f(e = f'(e) for all edges e in G except (u, v)
- f(u, v) = f'(u, x) (which equals f'(x, v) by flow conservation at (x))

Check capacity: f(u, v) = f'(u, x)<= c'(u, x) = c(u, v), so capacity satisfied.
Flow conservation in G at u and v is preserved because the net flow between u and v is the same as in G' :
In G' : flow from u to v = f'(u, x).
In G : flow from u to v = f(u, v) = f'(u, x) .
So f is valid in G with same value as f' .
Thus max-flow(G')<=max-flow(G) .

4. Edge cases

- If (u, v) is not used in some max flow of G ,then in G' we can se f'(u, x) = f'(x, v) = 0).
- The transformation preserves all s-t paths and their bottleneck capacities.

Yes, The maximum flow value in G' is the same as in G.

Example:
Graph (G) with nodes ({s,u,v,t}) and capacities:
c(s->u)=4; c(u->v)=3; c(v->t)=5; c(s->v)=2; c(u->t)=1.
One maximum flow in (G) is

- f(s->u)=4 split as f(u->v)=3; f(u->t)=1),
- f(s->v)=2; f(v->t)=3+2=5).

Flow value (=3+2+1=6). (No augmenting path remains.)

Form G' by splitting edge (u,v) into $u \xrightarrow{3} x \xrightarrow{3} v$ (both capacities (=3)). Use the flow

- f'(s->u)=4; f'(u->x)=3; f'(x->v)=3; f'(u->t)=1; f'(s->v)=2; f'(v->t)=5.)

This f' is feasible in G' (conservation holds at (x) since (3=3)) and has value (6). The same one-to-one mapping argument in the proof applies in both directions, so max-flow(G)=max-flow(G')=6).

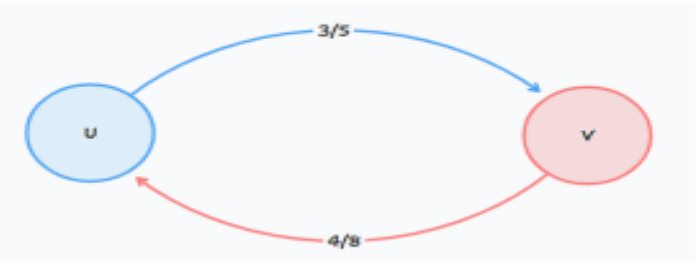
Lemma: Let (X, Y) be any s-t cut in a network. Given any flow f, the value of f is equal to the net flow across the cut, that is, $f(X, Y) = |f|$.

Proof: Recall that there are no edges leading into s, and so we have $|f| = f^{\text{out}}(s) = f^{\text{out}}(s) - f^{\text{in}}(s)$. Since all the other nodes of X must satisfy flow conservation it follows that

$$|f| = \sum_{x \in X} (f^{\text{out}}(x) - f^{\text{in}}(x))$$

Now, observe that every edge (u, v) where both u and v are in X contributes one positive term and one negative term of value $f(u, v)$ to the above sum, and so all of these cancel out. The only terms that remain are the edges that either go from X to Y (which contribute positively) and those from Y to X (which contribute negatively). Thus, it follows that the value of the sum is exactly $f(X, Y)$, and therefore $|f| = f(X, Y)$.

5. Given vertices u and v in a flow network, where capacity c(u, v) = 5 and c(v, u) = 8, suppose that 3 units of flow are shipped from u to v and 4 units are shipped from v to u. Compute the net flow from u to v by giving a diagram.



The net flow from vertex u to vertex v is **-1**, indicating that there is effectively a surplus of flow moving back from v to u.

Questions on Circulation and Extensions

15. Give the **reduction of the circulation problem** to a **network flow problem**.

David mount pdf(p-110)

16. In the context of **flow networks with upper and lower bounds**, explain how the feasibility condition can be checked using a flow model.

    David mount pdf(p-113)

17. Define the reduction of circulation problem to network flow problem.

A circulation problem can be reduced to a maximum flow problem by adding a super-source s and super-sink t.
For every node v, if its net demand is b(v), then add an edge s -> v with capacity b(v) when b(v)>0, and an edge v -> t) with capacity -b(v) when b(v)<0.
Finally, compute max-flow from s to t; a feasible circulation exists iff all edges out of s are saturated.

Example Circulation Problem

You have a directed graph with demands:

- Node **A** has demand **+3** (needs 3 units of inflow).
- Node **B** has demand **–3** (must send out 3 units).

Edges:

- **B → A** with capacity 5.

Goal: Check if a feasible circulation exists.

Step-by-Step Reduction to Max-Flow

1. Add a Super-Source (s)

For every node with **positive demand**:

- A has demand **+3 → add edge (s -> A)** with capacity 3.

2. Add a Super-Sink (t)

For every node with **negative demand**:

- B has demand **–3 → add edge (B -> t)** with capacity 3.

3. Keep Original Edges

- Keep B → A (capacity = 5).

Now the transformed graph is:

```
   (cap 3)     (cap 5)     (cap 3)
 s --------> A <----------- B --------> t
```

Run Max-Flow from s to t

Possible flow:

- s → A : 3
- A → B : 3 (uses part of available 5)
- B → t : 3

All edges from **s** are saturated ⇒ **feasible circulation exists**.

Final Conclusion

If max-flow saturates all edges outgoing from **s**, then the original circulation problem has a solution.
If not, no feasible circulation exists.

Question on Bipartite Matching and Reductions

18. Give a **polynomial-time reduction** algorithm from the **Maximum Bipartite Matching** problem to the **Maximum Flow** problem.

1. Problem definitions

- **Maximum Bipartite Matching**:
  Given bipartite graph G = (X ∪ Y, E), find a maximum cardinality set of edges M ⊆ E such that no two edges share a vertex.
- **Maximum Flow**:
  Given directed capacitated graph G' with source s and sink t, find a flow of maximum value.

2. Reduction construction

From G = (X ∪ Y, E) (undirected, bipartite):

1. Create a new directed graph G' :
   a. Keep all vertices X ∪ Y.
   b. Add a **source** s and a **sink** t.
2. **Edges and capacities**:
   a. Add directed edges from s to every vertex in X, each with capacity 1.
   b. For each original edge {x, y} ∈ E) with x ∈ X, y ∈ Y), add a directed edge x -> y with capacity 1.
   c. Add directed edges from every vertex in Y to t , each with capacity 1.

3. Why it works

- Each unit of flow from s to t corresponds to choosing one matching edge:
  - s -> x ensures x ∈ X is used at most once.
  - x -> y corresponds to selecting edge {x, y}.
  - (y \to t) ensures (y \in Y) is used at most once.
- Capacity 1 on all edges ensures that each x ∈ X and each y ∈ Y is matched at most once.
- Integer capacities ⇒ there exists an integer max flow (0/1 on edges), which corresponds directly to a matching.
- The size of the maximum matching = the value of the maximum flow.

4. Polynomial time

The reduction constructs G' in O(|V| + |E|) time:

- |V'| = |X| + |Y| + 2
- |E'| = |X| + |E| + |Y|

Max flow in G' can be found in polynomial time (e.g., Edmonds–Karp: (O(|V'| . |E'|^2))).

5. Example

Bipartite graph: X = {x_1, x_2}, Y = {y_1, y_2}
Edges: (x_1, y_1), (x_1, y_2), (x_2, y_1)
Flow network:
s -> x_1 (cap 1), s -> x_2 (cap 1)

x_1-> y_1) (cap 1), x_1-> y_2) (cap 1), x_2 -> y_1 (cap 1)
y_1-> t) (cap 1), y_2 -> t (cap 1)
Max flow = 2 (e.g., (x_1 -> y_2, x_2 -> y_1)), matching size = 2.
Construct flow network with source connected to X, X→Y edges from E, Y connected to sink, all capacities 1.

19. Define a **bipartite matching** problem and illustrate how it can be transformed into a flow network to find the maximum matching.
    David Mount pdf(p-110)
20. Define the general matching problem and bipartite matching problem.
    Then, give a polynomial-time reduction algorithm to reduce the bipartite matching problem to a network flow problem, justifying the time complexity.
    David Mount pdf(p-110)
21. Given a bipartite graph G = (U ∪ V, E) where
    U = {a, c, e, g, i} and V = {b, d, f, h,
    j},E={(a,d),(a,f),(b,c),(b,e),(c,d),(c,f),(c,h),(d,e),(d,g),(e,h),(f,g),(h,i),(i,j) find the maximum matching using the Hungarian Tree method.

We'll find the maximum bipartite matching using the Hungarian tree method (augmenting path search via BFS from unmatched vertices in (U)).

1. Initial matching
Let's start with an empty matching M = {}.
2. Step 1 — Find augmenting path from (a∈ U)
BFS tree from a:

- a unmatched.
- Neighbors: d, f (both free).
- Pick d: a-d is augmenting path.

Augment: M = { (a,d) }.
3. Step 2 — Find augmenting path from (b ∈ U)
b unmatched.
Neighbors: c, e (both free).
Pick c: b-c is augmenting path.
Augment: M = { (a,d), (b,c) } .
4. Step 3 — Find augmenting path from (e ∈ U)
e unmatched.
Neighbors: d, h.

- d is matched to a.
- h is free.

So path: e-h is augmenting.
Augment: M = { (a,d), (b,c), (e,h) }.

5. Step 4 — Find augmenting path from (g ∈ U)

g unmatched.
Neighbors: d, f.

- d matched to a.
- f free.

So path: g-f is augmenting.
Augment: M = { (a,d), (b,c), (e,h), (g,f) } .

6. Step 5 — Find augmenting path from (i ∈ U)

i unmatched.
Neighbors: h, j.

- h matched to e.
- j free.

So path: i-j is augmenting.
Augment: M = { (a,d), (b,c), (e,h), (g,f), (i,j) } .

7. Check

All vertices in U are matched:
(a ->d), (b->c), (e->h), (g ->f), (i -> j).
So maximum matching size = (5).
5 Matching edges: (a,d), (b,c), (e,h), (g,f), (i,j).

22. Prove or disprove: The Max-flow and Bipartite Matching problems are polynomial-time reducible to each other.

We need to check whether:

1. **Max-flow** ≤(_P) **Bipartite Matching**
2. **Bipartite Matching** ≤(_P) **Max-flow**

1. Bipartite Matching ≤(_P) Max-flow

This is **true** and well-known:

- Given a bipartite graph G = (U ∪ V, E), construct a flow network:
  - Source s connected to each u ∈ U with capacity 1.
  - Each original edge (u, v) directed u -> v with capacity 1.
  - Each v ∈V connected to sink t with capacity 1.
- The value of the maximum flow equals the size of the maximum matching.
- Reduction is polynomial (linear in graph size).

So **Bipartite Matching** is polynomial-time reducible to **Max-flow**.

2. Max-flow ≤(_P) Bipartite Matching

This would mean: given any flow network, we can construct a bipartite graph whose maximum matching size equals the max flow value.

This is **false** in general because:

- **Max-flow** is a more general problem: it can have arbitrary capacities, multiple paths, and is not restricted to bipartite structure or unit capacities.
- If such a reduction existed, we could solve general max-flow by solving bipartite matching, but bipartite matching is a special case of max-flow with capacities 1 and a certain structure.
- Known fact: Bipartite matching corresponds to max-flow in a unit-capacity, layered network. But general max-flow can have large capacities and complex cycles not expressible as a bipartite matching.

Counterexample idea:

Take a max-flow instance with source s, sink t, and one edge (s, t) with capacity $k > 1$.
Max flow = k.
A bipartite matching problem's matching size is at most $\min(|U|, |V|)$, but here k can be arbitrarily large independent of graph size — impossible to encode in a bipartite matching instance of size polynomial in original network unless we "simulate" capacities by copying nodes, but that would require k copies, which is not polynomial if k is large (pseudo-polynomial).
Thus, no general polynomial-time reduction from Max-flow to Bipartite Matching unless P = NP (which is open and unlikely).
Actually, known complexity:

- Bipartite Matching is in P.
- Max-flow is in P.

But a poly-time reduction from Max-flow to Bipartite Matching is not known and believed impossible because Bipartite Matching is a special case.

3. Conclusion

- **Bipartite Matching** → **Max-flow**: YES (standard construction).
- **Max-flow** → **Bipartite Matching**: NO, unless we allow pseudo-polynomial blow-up (copying edges capacity times), which is not polynomial-time.

Thus, they are **not** polynomial-time reducible to each other in **both directions**.

Disproved:Only one direction is polynomial-time reducible.

Questions on Applications and Complex Scenarios

23. In a hospital with *n* doctors and *m* patients, each doctor can treat a limited number of patients daily.

    Design an **algorithm using network flow principles** to maximize the number of treated patients and analyze its complexity.

Problem setup

- n doctors D_1, D_2, ……., D_n
- m patients P_1, P_2, ……, P_m
- Each doctor D_i can treat up to c_i patients.
- Each patient can be treated by some subset of doctors (given by eligibility list).

Goal → **maximize the number of patients treated**.

Algorithm (using Max-Flow)

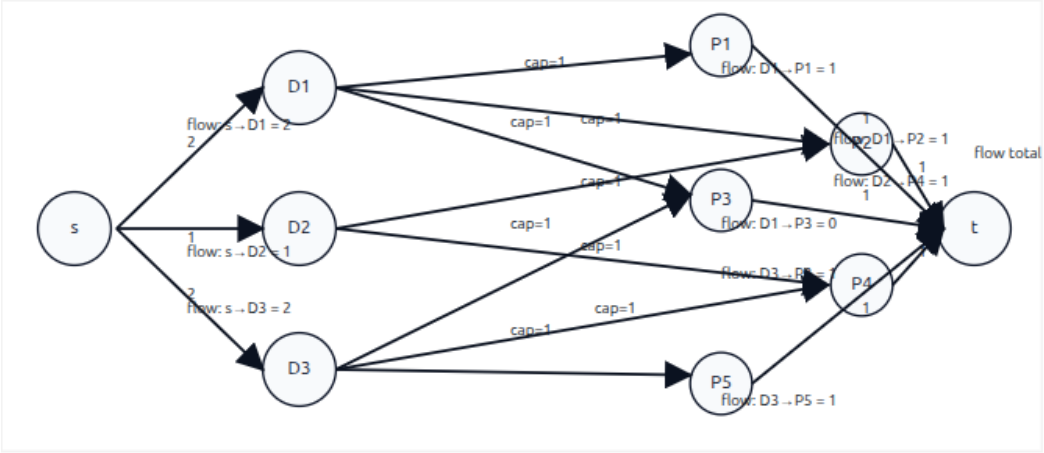
1. **Create a flow network G=(V,E) :**
   a. Add a **source** s and **sink** t .
   b. For each doctor D_i : add edge (s, D_i) with capacity c_i.
   c. For each patient P_j : add edge (P_j, t) with capacity 1 .
   d. For every eligible doctor–patient pair (D_i, P_j): add edge (D_i, P_j) with capacity 1 .
2. **Run a Max-Flow algorithm** (e.g., Edmonds–Karp or Dinic) on this network.
3. **Interpret the flow:**
   a. Each unit of flow from D_i ->P_j represents doctor D_i treating patient P_j.
   b. The total flow value = total number of treated patients (maximum possible).

Complexity analysis

- Graph has O(n + m) vertices and O(nm) edges in the worst case.
- Using **Edmonds–Karp**, runtime O(VE^2) = O((n+m)(nm)^2)) — high but polynomial.
- Using **Dinic's algorithm**, time (O(\sqrt{V}E) = O(\sqrt{n+m}·nm)).
- If all capacities are integers ≤ 1 (unit capacities), the special case runs in O(E \sqrt{V}) — efficient in practice.

✅ **Conclusion:**

Model doctors and patients as a bipartite flow network with capacities on doctor edges. The **maximum flow = maximum number of patients treated**, and the **time complexity** is polynomial — (O(nm)) to (O(E \sqrt{V})) depending on the max-flow algorithm used.



24. Construct an example where **circulations with lower bounds** are applied in an allocation or scheduling scenario such as survey or task assignment.
    David Mount pdf(p-115)
25. In a hospital, there are n doctors and two shifts (morning & evening).
    Each doctor can work in at most one shift per day and at most two days in total.
    Design a network flow model to assign doctors to shifts efficiently.

We can model this as a **maximum flow problem** in a suitably constructed flow network.

1. Define the problem in constraints

We have:

- n doctors D_1, D_2, ……, D_n
- 2 shifts per day: morning M, evening E
- Total days available: let's say k days (problem doesn't specify days, but we can generalize: suppose there are m days).
- Each doctor can work **at most one shift per day** (so cannot do both M and E on same day).
- Each doctor can work **at most two days in total** (over all days).

We want to **maximize the number of shift assignments** (or ensure some required number if given).

2. Flow network construction

We build a directed graph with capacities:

Nodes: Source s ,Sink t ,One node for each doctor: (d_1, …, d_n), One node for each shift on each day: e.g., (M_1, E_1, M_2, E_2, ……, M_m, E_m) if m days.

Edges:

1. s -> d_i: capacity 2 (each doctor works at most 2 days total; each day they can work 1 shift, so 2 capacity = 2 days max).
2. d_i -> M_j : capacity 1 (doctor i can work morning of day j at most once).
3. d_i -> E_j : capacity 1 (doctor i can work evening of day j at most once).
4. M_j -> t : capacity = number of doctors needed for morning shift on day j (or large if we just want to maximize).
5. E_j -> t): capacity = number of doctors needed for evening shift on day j (or large if maximizing).

3. Why it works

- Each unit of flow from s to t represents one doctor assigned to one shift on one day.
- Capacity 2 from s to d_i enforces "at most two days total" because each day the doctor can only work 1 shift (so 2 flow units = 2 days).
- The edges (d_i -> M_j) and (d_i -> E_j) have capacity 1, ensuring the doctor doesn't work both M and E on same day? Wait — careful: Actually, if a doctor works both M and E on same day, that would require 2 flow units from d_i to day j 's shifts. But that's not allowed by "at most one shift per day". So we must prevent that.

4. Preventing same-day both shifts

We can enforce "at most one shift per day per doctor" by **splitting each day-node**:

Better model:

Nodes: s, Doctor nodes (d_1, …. , d_n), For each day j , a "day-doctor" node day_{i,j}) for each doctor? That's too big. Instead:

Actually simpler:

We can keep the earlier model but add **day nodes** day_j for each day j:

- s -> d_i : cap 2
- d_i -> day_j : cap 1 (doctor works day j at most once — i.e., one shift)
- day_j -> M_j : cap = large (or required morning slots)
- day_j -> E_j : cap = large (or required evening slots)
- M_j -> t : cap = morning requirement
- E_j -> t : cap = evening requirement

But this still allows a doctor to be assigned to both M and E through day_j if capacities allow? No — because d_i -> day_j has capacity 1, so doctor can only be assigned to one shift on that day.
Yes, this works.

5. Final network structure

1. s -> d_i : capacity 2
2. d_i -> Day_j : capacity 1 for all i, j (if doctor can work that day; omit if not)
3. Day_j -> M_j : capacity = ∞ or morning requirement
4. Day_j -> E_j : capacity = ∞ or evening requirement
5. M_j -> t): capacity = morning requirement a_j
6. E_j -> t : capacity = evening requirement b_j

If we just want to maximize total shifts assigned, set a_j, b_j large enough (e.g., n).

6. Use case

Compute max flow from s to t .
If max flow = ∑ j (a_j + b_j) , all shifts filled.
Otherwise, some shifts unfilled due to doctor availability constraints.
Model as max flow with doctor nodes connected to day nodes (cap 1), day nodes to shifts, source to doctors (cap 2), shifts to sink as per requirements.

26. Given a scenario of resource allocation or scheduling with constraints, formulate the problem using a circulation network and verify feasibility using a flow transformation.
27. The following scenario is given: After a flood, food must be distributed to several locations using available road networks with limited capacities.
    Design a **flow network and algorithm** to decide if supplies meet demands, explaining whether total supply equals total demand.

We can model this as a **multi-source, multi-sink flow problem** and reduce it to a standard **single-source, single-sink max flow problem**.

1. Problem restatement

We have:

- **Supply nodes**: locations with supplies (e.g., warehouses) — each has a fixed supply amount S_i .

- **Demand nodes**: flood-affected locations needing food — each has a fixed demand D_j.
- **Road network**: edges between locations with **capacity** $c(u, v)$ = max food units that can be transported along that road per day.
- Question: Can we route supplies to meet **all demands** exactly?

Or: Does **total supply ≥ total demand** and can we distribute accordingly?

2. Flow network construction

We build a directed graph G' for max flow:

Nodes:

- Keep all original locations as nodes.
- Add a **super-source** s.
- Add a **super-sink** t.

Edges and capacities:

1. From s to each supply node i: capacity S_i (the total supply available at i).
2. Original road edges: directed or undirected?

If roads are two-way, we can put two directed edges each with the same capacity $c(u, v)$.

If one-way, one directed edge.

3. From each demand node j to t: capacity D_j (the demand at j).

3. Check feasibility condition

Let TotalSupply = $\sum_i S_i$

Let TotalDemand = $\sum_i D_j$

Necessary condition:

TotalSupply >= TotalDemand

Otherwise, impossible to meet all demands.

But this alone is **not sufficient** — the road network's capacities might limit flow even if total supply ≥ total demand.

4. Algorithm

1. Construct G' as above.
2. Compute **max flow** F from s to t in G'.
3. If F = TotalDemand, then **yes**, we can meet all demands.

If F < TotalDemand, then **no**, we cannot meet all demands (bottleneck in road capacities or supply locations' access to demand nodes).

5. Example

Suppose:

- Supply nodes: A (supply 50), B (supply 30)
- Demand nodes: X (demand 40), Y (demand 30)
- Roads: A -> X cap 30, A -> Y cap 20, B -> X cap 10, B -> Y cap 25.

TotalSupply = 80, TotalDemand = 70.

Build network:

- s -> A : cap 50
- s -> B : cap 30
- Existing roads with given capacities

- X -> t : cap 40
- Y -> t : cap 30

Compute max flow:We can check if it reaches 70.

6. Why total supply vs total demand matters

- If total supply < total demand → immediately "no".
- If total supply ≥ total demand, we still need to check max flow = total demand.
- If we just want to know if we can meet demands, compare max flow to total demand.
- If total supply > total demand, the extra supply is irrelevant — we only need to send total demand amount from s to t.

Final answer:

Use max flow with super-source connected to supplies (cap = supply), demands connected to super-sink (cap = demand), roads as edges with their capacities. Compare max flow value to total demand.