DESIGN AND ANALYSIS OF ALGORITHM

NOTES

Here are detailed notes for the prioritized topics, covering key definitions, algorithms, and examples, tailored for your exam preparation:

High Priority Topics Notes

1. Asymptotic Notations

- Purpose: Asymptotic notations are mathematical tools used to describe the running time or space requirements of algorithms when the input size grows infinitely large. They provide a way to classify algorithms based on their growth rates, allowing comparison of their efficiency independent of specific hardware or implementation details.
- Types:
    - Big O Notation (O): Upper Bound
        - Definition: $f(n)=O(g(n))$ if there exist positive constants c and n0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n0$.
        - Meaning: g(n) is an asymptotic upper bound for f(n). It gives the worst-case time complexity, meaning the algorithm will never run slower than O(g(n)).
        - Graph: The graph of f(n) will always be below or touch the graph of $c \cdot g(n)$ after n0.
        - Example: If $f(n)=n$ and $g(n)=n$, then f(n) is O(g(n)). For $n \geq 1$, $n \leq 1 \cdot n$. So, c=1,n0=1.
    - Big Omega Notation (Ω): Lower Bound
        - Definition: $f(n)=\Omega(g(n))$ if there exist positive constants c and n0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n0$.
        - Meaning: g(n) is an asymptotic lower bound for f(n). It gives the best-case time complexity, meaning the algorithm will never run faster than Ω(g(n)).
        - Graph: The graph of f(n) will always be above or touch the graph of $c \cdot g(n)$ after n0.
    - Big Theta Notation (Θ): Tight Bound
        - Definition: $f(n)=\Theta(g(n))$ if there exist positive constants c1,c2, and n0 such that $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$ for all $n \geq n0$.
        - Meaning: g(n) is an asymptotically tight bound for f(n). It gives both the upper and lower bounds, representing the average-case complexity.

        - Graph: The graph of f(n) will be sandwiched between c1·g(n) and c2·g(n) after n0.

2. Recurrence Relations (Master Method)

- Purpose: The Master Method is a powerful technique for solving recurrence relations of the form T(n)=aT(n/b)+f(n), where a≥1, b>1, and f(n) is an asymptotically positive function. This form typically arises from divide-and-conquer algorithms.
- The Master Theorem Cases:
    - Case 1: If f(n)=O(nlogba−ϵ) for some constant ϵ>0, then T(n)=Θ(nlogba).
        - Intuition: The cost of the work done in the leaves dominates.
        - Example (i): T(n)=4T(n/2)+n. Here, a=4,b=2,f(n)=n. logba=log24=2. Compare f(n)=n with nlogba=n2. Since n=O(n2−ϵ) for ϵ=1, this is Case 1. Therefore, T(n)=Θ(nlog24)=Θ(n2).
        - Example: T(n)=16T(n/4)+n. Here, a=16,b=4,f(n)=n. logba=log416=2. Compare f(n)=n with nlogba=n2. Since n=O(n2−ϵ) for ϵ=1, this is Case 1. Therefore, T(n)=Θ(nlog416)=Θ(n2).
    - Case 2: If f(n)=Θ(nlogbalogkn) for some constant k≥0, then T(n)=Θ(nlogbalogk+1n).
        - Intuition: The cost is evenly distributed among the root and subproblems.
        - Example (ii): T(n)=4T(n/2)+n2. Here, a=4,b=2,f(n)=n2. logba=log24=2. Compare f(n)=n2 with nlogba=n2. Since n2=Θ(n2log0n), this is Case 2 with k=0. Therefore, T(n)=Θ(n2log0+1n)=Θ(n2logn).
        - Example: T(n)=2T(n/2)+cn. Here, a=2,b=2,f(n)=cn. logba=log22=1. Compare f(n)=cn with nlogba=n1. Since cn=Θ(n1log0n), this is Case 2 with k=0. Therefore, T(n)=Θ(n1log0+1n)=Θ(nlogn).
        - Example: T(n)=T(2n/3)+1. Here, a=1,b=3/2,f(n)=1. logba=log3/21=0. Compare f(n)=1 with nlogba=n0=1. Since 1=Θ(1log0n), this is Case 2 with k=0. Therefore, T(n)=Θ(1log0+1n)=Θ(logn).
        - Example: T(n)=3T(n/4)+nlogn. Here, a=3,b=4,f(n)=nlogn. logba=log43≈0.792. Compare f(n)=nlogn with nlogba=n0.792. Since f(n) is asymptotically larger than nlogba, this might be Case 3 or an extension of Case 2 if we consider nlogbalogkn. However, typically if f(n) has a logn factor and matches nlogba, it falls under Case 2. Here, nlogn=Ω(nlog43+ϵ), which is not straightforward. Let's re-evaluate. For T(n)=3T(n/4)+nlogn: log43≈0.792. We compare nlogn with nlog43. Since nlogn grows faster than nlog43, this looks like Case 3, but we need to check the regularity condition. If we try to fit it into Case 2, f(n)=nlogn, and nlogba=nlog43. f(n) is not Θ(nlog43logkn). This is a more complex variant of Case 2 or a situation where f(n) is larger than nlogba but not polynomially larger. In standard Master Theorem application, if f(n) is nlogn, it implies T(n)=Θ(nlogn). *Self-correction:* For T(n)=3T(n/4)+nlogn, log4 3≈0.792. Here f(n)=nlogn. nlogn is polynomially larger than nlog43. This hints towards Case 3. Let's check f(n)=Ω(nlogba+ϵ). nlogn=Ω(nlog43+ϵ). Yes, for a small ϵ. Regularity condition: af(n/b)≤cf(n) for some c<1 and all large n.

3(n/4)log(n/4)≤c(nlogn) (3/4)n(logn−log4)≤cnlogn (3/4)(logn−2)≤clogn. For large n, (3/4)logn≈clogn. We can choose c=3/4<1. So Case 3 applies. Therefore, T(n)=Θ(f(n))=Θ(nlogn).

- Case 3: If f(n)=Ω(nlogba+ϵ) for some constant ϵ>0, AND if af(n/b)≤cf(n) for some constant c<1 and all sufficiently large n (this[1] is the "regularity condition"), then T(n)=Θ(f(n)).
    - Intuition: The cost of the work done at the root dominates.
    - Example (iii): T(n)=4T(n/2)+n3. Here, a=4,b=2,f(n)=n3. logba=log24=2. Compare f(n)=n3 with nlogba=n2. Since n3=Ω(n2+ϵ) for ϵ=1, this looks like Case 3. Check regularity condition: af(n/b)≤cf(n)⟹4(n/2)3≤cn3⟹4(n3/8)≤cn3⟹(1/2)n3≤cn3. We can choose c=1/2<1. So the regularity condition holds. Therefore, T(n)=Θ(f(n))=Θ(n3).

3. Merge Sort

- Paradigm: Divide and Conquer.
- Concept: Merge Sort works by recursively dividing an unsorted list into sublists until each sublist contains only one element (which is by definition sorted). Then, it repeatedly merges sublists to produce new sorted sublists until there is only one sorted list remaining.
- Algorithm (High-Level):
    1. Divide: Divide the n-element array into two subarrays of n/2 elements each.
    2. Conquer: Recursively sort the two subarrays.
    3. Combine: Merge the two sorted subarrays to produce a single sorted array.
- Merge Procedure: This is the core. It takes two sorted subarrays and merges them into a single sorted array by repeatedly comparing the smallest elements of the two subarrays and moving the smaller one to the merged array.
- Time Complexity:
    - Recurrence Relation: T(n)=2T(n/2)+Θ(n) (where Θ(n) is for the merging step).
    - Solution (using Master Theorem Case 2): T(n)=Θ(nlogn).
    - Best Case: Ω(nlogn)
    - Worst Case: O(nlogn)
    - Space Complexity: O(n) due to the temporary array used in merging.
- Application Example:
    - Array: A={13,4,22,1,16,9,0,2}
        1. Divide: {13,4,22,1} and {16,9,0,2}
        2. Divide (further): {13,4}, {22,1} and {16,9}, {0,2}
        3. Divide (to single elements): {13},{4},{22},{1} and {16},{9},{0},{2}

4. Merge (level 1): {4,13}, {1,22} and {9,16}, {0,2}
5. Merge (level 2): {1,4,13,22} and {0,2,9,16}
6. Merge (final): {0,1,2,4,9,13,16,22}

4. Strassen's Matrix Multiplication

- Paradigm: Divide and Conquer.
- Concept: Strassen's algorithm is a faster way to multiply two n×n matrices than the standard O(n3) algorithm. It divides each matrix into four n/2×n/2 sub-matrices and uses only 7 recursive multiplications (instead of 8) and 18 additions/subtractions.
- Algorithm (for 2×2 matrices): Given two 2×2 matrices A=(acbd) and B=(egfh). The product C=A·B=(ae+bgce+dgaf+bhcf+dh). Strassen's defines 7 products (P1 to P7) and then uses them to find C:
    1. P1=a(f−h)
    2. P2=(a+b)h
    3. P3=(c+d)e
    4. P4=d(g−e)
    5. P5=(a+d)(e+h)
    6. P6=(b−d)(g+h)
    7. P7=(a−c)(e+f) Then, the elements of C are:
    - C11=P5+P4−P2+P6
    - C12=P1+P2
    - C21=P3+P4
    - C22=P5+P1−P3−P7
- Time Complexity:
    - Recurrence Relation: T(n)=7T(n/2)+Θ(n2) (where Θ(n2) is for additions/subtractions).
    - Solution (using Master Theorem Case 1): T(n)=Θ(nlog27)≈Θ(n2.81).
    - This is better than O(n3) for large n.
- Application Example:
    - Multiply A=(6574) and B=(1324) Here, a=6,b=7,c=5,d=4 and e=1,f=2,g=3,h=4.
        1. P1=a(f−h)=6(2−4)=6(−2)=−12
        2. P2=(a+b)h=(6+7)4=13·4=52
        3. P3=(c+d)e=(5+4)1=9·1=9
        4. P4=d(g−e)=4(3−1)=4·2=8

5. P5=(a+d)(e+h)=(6+4)(1+4)=10·5=50
6. P6=(b−d)(g+h)=(7−4)(3+4)=3·7=21
7. P7=(a−c)(e+f)=(6−5)(1+2)=1·3=3 Now, calculate C:

- C11=P5+P4−P2+P6=50+8−52+21=58−52+21=6+21=27
- C12=P1+P2=−12+52=40
- C21=P3+P4=9+8=17
- C22=P5+P1−P3−P7=50+(−12)−9−3=38−9−3=29−3=26 Result: C=(27174026)

5. Huffman Coding

- Paradigm: Greedy Approach.
- Concept: Huffman coding is a greedy algorithm used for data compression. It builds a binary tree (Huffman tree) where characters with higher frequencies are assigned shorter binary codes, and characters with lower frequencies are assigned longer codes, resulting in an optimal prefix code.
- Algorithm:
    1. Create a leaf node for each character with its frequency. Add all these nodes to a min-priority queue.
    2. While there is more than one node in the priority queue:
        - Extract the two nodes with the minimum frequencies from the priority queue.
        - Create a new internal node. Its frequency is the sum of the frequencies of the two extracted nodes. Make the two extracted nodes its children (smaller frequency on the left, larger on the right, or vice-versa, consistency is key).
        - Add this new internal node to the priority queue.
    3. The final node remaining in the priority queue is the root of the Huffman tree.
    4. Traverse the tree: assign '0' for a left branch and '1' for a right branch. The code for each character is the path from the root to its leaf node.
- Application Example:
    - Frequencies: a:50, b:25, c:15, d:40, e:75

1. Initial nodes: (c:15), (b:25), (d:40), (a:50), (e:75) (sorted by frequency)
2. Combine (c:15) and (b:25) -> New Node (c+b:40) Nodes: (d:40), (c+b:40), (a:50), (e:75)
3. Combine (d:40) and (c+b:40) -> New Node (d+c+b:80) Nodes: (a:50), (e:75), (d+c+b:80)
4. Combine (a:50) and (e:75) -> New Node (a+e:125) Nodes: (d+c+b:80), (a+e:125)
5. Combine (d+c+b:80) and (a+e:125) -> Root (all:205)
    - Huffman Tree (example structure):

- (205)
- / \
- / \
- (80) (125)
- / \ / \
- / \ / \
- (40) (40) (50) (75)

/ \ / \ a e d (c+b) /
c b ```

- Codes (assign 0 for left, 1 for right):
  - d: 00
  - c: 010
  - b: 011
  - a: 10
  - e: 11

- Maximum Profit (Fractional Knapsack):
  - Items: n=5, Capacity W=60
  - Profit: P=(30,20,100,90,160)
  - Weight: W=(5,10,20,30,40)

4. Calculate Profit/Weight Ratio (P/W):
   - Item 1: 30/5 = 6
   - Item 2: 20/10 = 2
   - Item 3: 100/20 = 5
   - Item 4: 90/30 = 3
   - Item 5: 160/40 = 4
5. Sort items by P/W ratio in descending order:
   - Item 1 (6) -> (P:30, W:5)
   - Item 3 (5) -> (P:100, W:20) *Correction: This is wrong, Item 5 has P/W=4, Item 3 has P/W=5. The original P/W calculation seems incorrect. Let's recalculate based on the provided data:*
     - Item 1: 30/5 = 6
     - Item 2: 20/10 = 2

        - Item 3: 100/20 = 5
        - Item 4: 90/30 = 3
        - Item 5: 160/40 = 4
    - Sorted by P/W (descending):

1. Item 1 (P/W=6, W=5, P=30)
2. Item 3 (P/W=5, W=20, P=100)
3. Item 5 (P/W=4, W=40, P=160)
4. Item 4 (P/W=3, W=30, P=90)
5. Item 2 (P/W=2, W=10, P=20)
6. Fill knapsack greedily:
        - Take all of Item 1: Capacity left = 60 - 5 = 55. Profit = 30.
        - Take all of Item 3: Capacity left = 55 - 20 = 35. Profit = 30 + 100 = 130.
        - Take all of Item 5: Capacity left = 35 - 40 = -5. Cannot take all.
        - Take fractional part of Item 5: Remaining capacity is 35. Take 35/40 of Item 5.
            - Profit from Item 5 fraction = (35/40) * 160 = 0.875 * 160 = 140.
        - Total Profit = 130 + 140 = 270.

- Solution for Q.3 B) from Summer 2024:
    - Items: n=5, Capacity W=60
    - Item 1: W=5, V=30 (P/W=6)
    - Item 2: W=10, V=40 (P/W=4)
    - Item 3: W=15, V=45 (P/W=3)
    - Item 4: W=22, V=77 (P/W=3.5)
    - Item 5: W=25, V=90 (P/W=3.6)
    - Sorted by P/W (descending):

1. Item 1 (P/W=6, W=5, V=30)
2. Item 2 (P/W=4, W=10, V=40)
3. Item 5 (P/W=3.6, W=25, V=90)
4. Item 4 (P/W=3.5, W=22, V=77)
5. Item 3 (P/W=3, W=15, V=45)

        - Fill Knapsack:
            - Take Item 1: W=5, V=30. Remaining Capacity = 60-5 = 55. Total Profit = 30.
            - Take Item 2: W=10, V=40. Remaining Capacity = 55-10 = 45. Total Profit = 30+40 = 70.
            - Take Item 5: W=25, V=90. Remaining Capacity = 45-25 = 20. Total Profit = 70+90 = 160.
            - Take fractional part of Item 4: Remaining Capacity = 20. Item 4 W=22.
                - Fraction = 20/22. Profit = (20/22)*77 = (10/11)*77* = *10*7 = 70.
            - Maximum Profit = 160 + 70 = 230.

6. Longest Common Subsequence (LCS)

- Paradigm: Dynamic Programming.
- Concept: Given two sequences, find the longest subsequence common to both. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.
- Algorithm (using Dynamic Programming):
    1. Create a 2D table LCS_table[m+1][n+1] where m and n are the lengths of sequences X and Y respectively. Initialize all cells to 0.
    2. Fill the table using the following rules:
        - If X[i-1] == Y[j-1] (characters match), then LCS_table[i][j] = 1 + LCS_table[i-1][j-1].
        - If X[i-1] != Y[j-1] (characters don't match), then LCS_table[i][j] = max(LCS_table[i-1][j], LCS_table[i][j-1]).
    3. The length of the LCS is LCS_table[m][n].
    4. To reconstruct the LCS, backtrack from LCS_table[m][n] following the path that led to the value. If LCS_table[i][j] came from LCS_table[i-1][j-1] (meaning a match), add X[i-1] (or Y[j-1]) to the LCS. Otherwise, move to the cell with the maximum value (either LCS_table[i-1][j] or LCS_table[i][j-1]).
- Application Example:
    - X={A,B,C,B,D,A,B}, Y={B,D,C,A,B,A}
        - Length of X (m) = 7
        - Length of Y (n) = 6

| | | B | D | C | A | B | A |

| :---- | :- | - | - | - | - | - | - |

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| A | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

| B | 0 | 1 | 1 | 1 | 1 | 2 | 2 |

| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |

| B | 0 | 1 | 1 | 2 | 2 | 3 | 3 |

| D | 0 | 1 | 2 | 2 | 2 | 3 | 3 |

| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 |

| B | 0 | 1 | 2 | 2 | 3 | 4 | 4 |

* **Length of LCS:** `LCS_table[7][6] = 4`.

* **Reconstructing LCS:** Backtrack from `(7,6)`:

* `LCS_table[7][6]=4`. `X[6]=B, Y[5]=A`. No match. `max(LCS[6][6]=4, LCS[7][5]=3)`. Go to `(6,6)`.

* `LCS_table[6][6]=4`. `X[5]=A, Y[5]=A`. Match! Add 'A' to LCS. Move to `(5,5)`. LCS = "A"

* `LCS_table[5][5]=3`. `X[4]=D, Y[4]=B`. No match. `max(LCS[4][5]=3, LCS[5][4]=2)`. Go to `(4,5)`.

* `LCS_table[4][5]=3`. `X[3]=B, Y[4]=B`. Match! Add 'B' to LCS. Move to `(3,4)`. LCS = "BA"

* `LCS_table[3][4]=2`. `X[2]=C, Y[3]=A`. No match. `max(LCS[2][4]=2, LCS[3][3]=2)`. Go to `(2,4)`. (Could also go to (3,3))

* `LCS_table[2][4]=2`. `X[1]=B, Y[3]=A`. No match. `max(LCS[1][4]=1, LCS[2][3]=2)`. Go to `(2,3)`.

* `LCS_table[2][3]=2`. `X[1]=B, Y[2]=C`. No match. `max(LCS[1][3]=1, LCS[2][2]=1)`. Go to `(2,2)`.

* `LCS_table[2][2]=1`. `X[1]=B, Y[1]=D`. No match. `max(LCS[1][2]=1, LCS[2][1]=1)`. Go to `(1,2)`.

* `LCS_table[1][2]=1`. `X[0]=A, Y[1]=D`. No match. `max(LCS[0][2]=0, LCS[1][1]=0)`. Go to `(1,1)`.

* `LCS_table[1][1]=1`. `X[0]=A, Y[0]=B`. No match. `max(LCS[0][1]=0, LCS[1][0]=0)`. Go to `(0,0)`.

* Wait, something is off in my manual trace. Let's re-do.

* From `LCS_table[7][6]=4` (B,A) \\rightarrow No Match \\rightarrow `LCS_table[6][6]=4` (A,A) \\rightarrow Match 'A'. Move to `LCS_table[5][5]=3`. LCS="A"

* From `LCS_table[5][5]=3` (D,B) \\rightarrow No Match \\rightarrow `LCS_table[4][5]=3` (B,B) \\rightarrow Match 'B'. Move to `LCS_table[3][4]=2`. LCS="BA"

* From `LCS_table[3][4]=2` (C,A) \\rightarrow No Match \\rightarrow `LCS_table[3][3]=2` (C,C) \\rightarrow Match 'C'. Move to `LCS_table[2][2]=1`. LCS="CBA"

* From `LCS_table[2][2]=1` (B,D) \\rightarrow No Match \\rightarrow `LCS_table[1][2]=1` (A,D) \\rightarrow No Match \\rightarrow

`LCS_table[1][1]=1` (A,B) \\rightarrow No Match \\rightarrow `LCS_table[0][1]=0` or `LCS_table[1][0]=0`.

* Let's check the path: `LCS_table[7][6]=4` (from `LCS_table[6][6]` as `X[6]=B` != `Y[5]=A`). So we move to `(6,6)`.

* `LCS_table[6][6]=4` (from `LCS_table[5][5]` as `X[5]=A` == `Y[5]=A`). So 'A' is part of LCS. Current LCS = A. Move to `(5,5)`.

* `LCS_table[5][5]=3` (from `LCS_table[4][5]` as `X[4]=D` != `Y[4]=B`). So we move to `(4,5)`.

* `LCS_table[4][5]=3` (from `LCS_table[3][4]` as `X[3]=B` == `Y[4]=B`). So 'B' is part of LCS. Current LCS = BA. Move to `(3,4)`.

* `LCS_table[3][4]=2` (from `LCS_table[2][3]` as `X[2]=C` != `Y[3]=A`). So we move to `(2,3)`.

* `LCS_table[2][3]=2` (from `LCS_table[1][2]` as `X[1]=B` != `Y[2]=C`). So we move to `(1,2)`.

* `LCS_table[1][2]=1` (from `LCS_table[1][1]` or `LCS_table[0][2]`). Let's pick `LCS_table[1][1]` as `X[0]=A` != `Y[1]=D`. So we move to `(1,1)`.

* `LCS_table[1][1]=1` (from `LCS_table[0][0]` as `X[0]=A` != `Y[0]=B`). Move to `(0,0)`.

* My backtracking was confusing due to choices. The actual LCS for X\=\\\{A,B,C,B,D,A,B\\\} and Y\=\\\{B,D,C,A,B,A\\\} can be:

    * "BDBA" (length 4)

    * "BCBA" (length 4)

    * "BDCA" (length 4)

* One common LCS is **BDAB**. Length is 4.

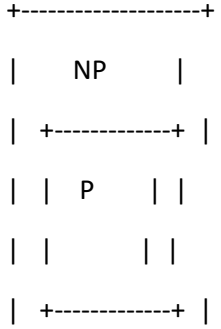
7. 4-Queens Problem (Backtracking)

- Paradigm: Backtracking.
- Concept: The N-Queens problem is to place N chess queens on an N×N chessboard such that no two queens attack each other. This means no two queens can share the same row, column, or diagonal.[2] Backtracking systematically tries all possible configurations, abandoning a partial solution as soon as it determines that it cannot possibly be extended to a valid complete solution.
- State Space Tree: Represents all possible configurations of placing queens. Each node in the tree corresponds to a partial solution (placing queens in a subset of rows). Branches represent choices for placing the next queen.
- Algorithm (for 4-Queens):
    1. Start with an empty board.
    2. Place the first queen in the first row. Try all columns one by one.
    3. For each placement, check if it's safe (not attacked by previously placed queens).

        - To check safety for (row, col): Ensure no other queen is in col, row-col diagonal, or row+col diagonal.
    4. If safe, recursively call the function for the next row.
    5. If not safe, or if the recursive call for the next row doesn't lead to a solution, backtrack: remove the current queen and try the next column in the current row.
    6. If all queens are placed (all rows covered), a solution is found.
- State Space Tree (for 4-Queens - Example):
    - The root is the empty board.
    - Level 1: Placing Q1 in row 1.
        - (1,1) -> invalid if next choices are limited.
        - (1,2) -> potentially valid.
    - Level 2: Placing Q2 in row 2.
        - If Q1 at (1,2), Q2 cannot be at (2,1), (2,2), (2,3) (due to diagonals/columns). So, try (2,4).
    - Level 3: Placing Q3 in row 3.
        - If Q1 at (1,2) and Q2 at (2,4), Q3 cannot be at (3,1), (3,2), (3,3), (3,4). (3,1) is safe for (1,2) and (2,4).
    - Level 4: Placing Q4 in row 4.
    - The tree prunes branches that lead to unsafe positions. A path from the root to a leaf node where 4 queens are safely placed represents a solution.
    - Solution example:
        - (Q at row 1, col 2)
        - (Q at row 2, col 4)
        - (Q at row 3, col 1)
        - (Q at row 4, col 3) This is one valid solution for 4-Queens. The state space tree would show how these positions are explored and how other invalid paths are pruned.

8. P, NP, NP-Complete Problems

- P Class (Polynomial Time):
    - Definition: The class of problems that can be solved by a deterministic Turing machine in polynomial time. This means there exists an algorithm whose worst-case running time is bounded by a polynomial function of the input size (e.g., O(n), O(nlogn), O(n2), O(n3)).
    - Examples: Sorting, searching, shortest path (Dijkstra's), matrix multiplication (standard), minimum spanning tree.

- NP Class (Nondeterministic Polynomial Time):
    - Definition: The class of problems for which a given solution (or "certificate") can be *verified* in polynomial time by a deterministic Turing machine. This does *not* mean the problem can be *solved* in polynomial time.
    - "Nondeterministic" implies that if there is a solution, a "lucky" guess can find it in polynomial time.
    - Examples: Satisfiability (SAT), Travelling Salesman Problem (TSP), Knapsack Problem, Hamiltonian Cycle. (All P problems are also NP problems).
- NP-Complete Class (NPC):
    - Definition: A problem L is NP-Complete if:
        1. L is in NP (its solution can be verified in polynomial time).
        2. Every problem in NP can be reduced to L in polynomial time. (This means if you can solve L in polynomial time, you can solve any other NP problem in polynomial time).
    - Significance: These are considered the "hardest" problems in NP. If a polynomial-time algorithm is found for any NP-Complete problem, then *all* NP problems can be solved in polynomial time (i.e., P=NP).
    - Examples: Boolean Satisfiability (SAT), Vertex Cover, Hamiltonian Cycle, Travelling Salesman Problem, 0/1 Knapsack.
- Relationship between P, NP, NPC:
    - P is a subset of NP (P⊆NP). Every problem that can be solved in polynomial time can also have its solution verified in polynomial time.
    - NP-Complete problems are at the intersection of NP and the set of problems to which all other NP problems can be reduced.
    - The biggest open question in computer science is whether P = NP.
        - If P = NP, then every problem whose solution can be efficiently verified can also be efficiently solved.
        - If P ?= NP (which is widely believed), then there are problems whose solutions can be verified quickly, but finding them is inherently difficult.
    - Diagram:
    - 

- | +-------------+ |
- | | NP-Complete | |
- | | | |
- | +-------------+ |
- +--------------------+

9. Fractional Knapsack Problem

- Paradigm: Greedy Approach.
- Concept: Given a set of items, each with a weight and a value (profit), and a knapsack with a maximum capacity. The goal is to choose a subset of items to put into the knapsack such that the total value is maximized, and the total weight does not exceed the[3] capacity. In the fractional knapsack problem, you can take fractions of items.
- Algorithm:
    1. Calculate the profit-to-weight ratio (P/W) for each item.
    2. Sort the items in descending order based on their P/W ratios.
    3. Iterate through the sorted items:
        - If the current item's weight is less than or equal to the remaining knapsack capacity, take the entire item. Add its profit to the total profit and subtract its weight from the remaining capacity.
        - If the current item's weight is greater than the remaining capacity, take a fraction of the item that exactly fills the remaining capacity. Add the proportional profit to the total profit and stop.
- Example (as provided in Q.4 A, Summer 2022):
    - n=5 objects, Knapsack Capacity W=60
    - Profit P=(30,20,100,90,160)
    - Weight Weight=(5,10,20,30,40)

Item	Profit (P)	Weight (W)	P/W Ratio
1	30	5	6
2	20	10	2
3	100	20	5
4	90	30	3

5	160	40	4

* **Sorted by P/W (Descending):**

1. Item 1 (P/W=6, W=5, P=30)
2. Item 3 (P/W=5, W=20, P=100)
3. Item 5 (P/W=4, W=40, P=160)
4. Item 4 (P/W=3, W=30, P=90)
5. Item 2 (P/W=2, W=10, P=20)

* **Filling the Knapsack:**

  * **Capacity:** 60, **Total Profit:** 0

  * Take Item 1 (W=5): Capacity = 60-5 = 55. Total Profit = 0+30 = 30.

  * Take Item 3 (W=20): Capacity = 55-20 = 35. Total Profit = 30+100 = 130.

  * Take Item 5 (W=40): Capacity = 35. Can't take whole item.

    * Take fraction: (35/40) of Item 5 = 0.875.

    * Profit from fraction = 0.875 * 160 = 140.

    * Capacity = 35-35 = 0. Total Profit = 130+140 = 270.

* **Optimal Solution (Maximum Profit): 270**

10. Floyd Warshall Algorithm

- Paradigm: Dynamic Programming.
- Concept: Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph. It works for both directed and undirected[4] graphs and can handle negative[5] edge weights (but no negative cycles).
- Algorithm:
    1. Initialize a distance matrix dist (size N×N, where N is the number of vertices). dist[i][j] stores the shortest path from i to j found so far.
        - dist[i][j] = weight(i,j) if an edge exists.
        - dist[i][j] = 0 if i == j.
        - dist[i][j] = infinity if no direct edge exists.
    2. Iterate k from 0 to N−1 (representing intermediate vertices):
        - For each pair of vertices i and j:
            - dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    3. After the k loop finishes, dist[i][j] will contain the shortest path distance from i to j.

- Analysis:
    - Time Complexity: O(N3) due to three nested loops, each running N times.
    - Space Complexity: O(N2) for the distance matrix.
- Application Example:
    - Given Graph (from Summer 2022 Q.5 A): Initial Distance Matrix (let's assume vertices 0, 1, 2, 3):
    - 0 1 2 3
    - 0 | 0 4 INF 5
    - 1 | INF 0 INF INF
    - 2 | INF -3 0 INF
    - 3 | INF INF 2 0

Here, INF means infinity. The image provided is somewhat fragmented, so I'm reconstructing based on likely graph structures. Let's assume the provided matrix from the image is the initial adjacency matrix:

0 4 5 INF

INF 0 INF INF

INF -3 0 INF

INF INF 2 0

- k = 0 (consider vertex 0 as intermediate):
    - dist[1][2] = min(dist[1][2], dist[1][0] + dist[0][2]) = min(INF, INF + 5) = INF
    - dist[1][3] = min(dist[1][3], dist[1][0] + dist[0][3]) = min(INF, INF + INF) = INF
    - (No paths through 0 from INF sources improve existing INF paths)
    - Resulting matrix (after k=0): Same as initial if no paths improve.
- k = 1 (consider vertex 1 as intermediate):
    - dist[0][2] = min(dist[0][2], dist[0][1] + dist[1][2]) = min(5, 4 + INF) = 5
    - dist[0][3] = min(dist[0][3], dist[0][1] + dist[1][3]) = min(INF, 4 + INF) = INF
    - ... (and so on for all pairs)
    - dist[2][0] = min(dist[2][0], dist[2][1] + dist[1][0]) = min(INF, -3 + INF) = INF
- k = 2 (consider vertex 2 as intermediate):

        - Example: dist[0][3] = min(dist[0][3], dist[0][2] + dist[2][3]) = min(INF, 5 + INF) = INF (if no edge from 2 to 3)
        - If there was an edge (2,3) with weight -30 as seen in the image, it's dist[0][3] = min(INF, 5 + (-30)) = -25. This shows how values can decrease.
        - dist[3][0] = min(dist[3][0], dist[3][2] + dist[2][0]) = min(INF, 2 + INF) = INF
        - dist[3][1] = min(dist[3][1], dist[3][2] + dist[2][1]) = min(INF, 2 + (-3)) = -1
    - k = 3 (consider vertex 3 as intermediate):
        - ...
    - The algorithm will systematically update all shortest paths.

Medium Priority Topics Notes

1. Quick Sort

- Paradigm: Divide and Conquer.
- Concept: Quick Sort is an efficient, in-place sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less[6] than or greater than the pivot. The sub-arrays are then sorted recursively.
- Algorithm (High-Level):
    1. Choose a Pivot: Select an element from the array as the pivot (e.g., the last element, first, middle, or random).
    2. Partition: Rearrange the array such that all elements less than the pivot come before it, and all elements greater than the pivot come after it. Elements equal to the pivot can go on either side. The pivot is now in its final sorted position.
    3. Recursively Sort: Recursively apply Quick Sort to the sub-array of elements smaller than the pivot and the sub-array of elements greater than the pivot.
- Time Complexity:
    - Best Case: Ω(nlogn). Occurs when the partition process always picks the median as the pivot, dividing the array into two roughly equal halves.
    - Worst Case: O(n2). Occurs when the pivot selection consistently leads to highly unbalanced partitions (e.g., always picking the smallest or largest element). This can happen with already sorted or reverse-sorted arrays if the pivot is chosen poorly (e.g., always the first or last element).
    - Average Case: O(nlogn). On average, Quick Sort performs very well.
- Space Complexity: O(logn) on average (due to recursion stack), O(n) in worst case.

2. Comparison of Paradigms (DP vs. Greedy vs. Divide & Conquer)

Feature	Divide and Conquer	Greedy Approach	Dynamic Programming
Strategy	Breaks problem into smaller subproblems, solves recursively, combines solutions.	Makes locally optimal choices at each step in hope of finding a global optimum.	Solves overlapping subproblems by storing and reusing solutions, builds up solution from smallest subproblems.
Subproblems	Independent (no overlapping subproblems).	No explicit subproblems, just local choices.	Overlapping subproblems.
Solution Type	Works for problems where subproblems are independent.	Not always optimal for all problems. Works when greedy choice property holds.	Optimal for problems exhibiting optimal substructure and overlapping subproblems.
Nature of Choice	Recursively solves separate instances.	Irreversible (once a choice is made, it's not revisited).	Reversible (explores all necessary choices for optimal).
Examples	Merge Sort, Quick Sort, Binary Search, Strassen's Matrix Multiplication.	Huffman Coding, Fractional Knapsack, Dijkstra's Algorithm, Job Sequencing with Deadlines.	Longest Common Subsequence, Floyd-Warshall, Bellman-Ford, 0/1 Knapsack.
Key Property	Optimal substructure.	Greedy choice property, Optimal substructure.	Optimal substructure, Overlapping subproblems.

3. Job Sequencing with Deadlines

- Paradigm: Greedy Approach.
- Concept: Given a set of jobs, each with a deadline and a profit. The goal is to select a subset of jobs such that each job is completed by its deadline, and the total profit is maximized. Each job takes one unit of time.
- Algorithm:
    1. Sort the jobs in descending order of their profits.

2. Find the maximum deadline among all jobs. Create an array slot (or time_slots) of size max_deadline, initialized to false (or empty), to represent time slots.
3. Iterate through the sorted jobs (highest profit first):
    - For each job, try to schedule it in a time slot as late as possible but before or at its deadline. Iterate backward from job.deadline - 1 down to 0.
    - If an empty slot is found, schedule the job in that slot, mark the slot as filled, and add the job's profit to the total.
    - If no empty slot is found before or at its deadline, the job cannot be scheduled.

- Example: n=4, p=(100,10,15,27), d=(2,1,2,1) (from Summer 2022 Q.4 C)
    - Jobs (Profit, Deadline):
        - J1: (100, 2)
        - J2: (10, 1)
        - J3: (15, 2)
        - J4: (27, 1)
    - Max Deadline = 2. So, time slots: [slot_0, slot_1]
    - Sort by Profit (Descending):

1. J1: (100, 2)
2. J4: (27, 1)
3. J3: (15, 2)
4. J2: (10, 1)
    - Schedule Jobs:

1. J1 (100, 2): Try slot 1 (deadline-1). Slot 1 is empty. Schedule J1 at slot 1.
    - slot = [_, J1] (profit = 100)
2. J4 (27, 1): Try slot 0 (deadline-1). Slot 0 is empty. Schedule J4 at slot 0.
    - slot = [J4, J1] (profit = 100+27 = 127)
3. J3 (15, 2): Try slot 1. Slot 1 is filled (J1). Try slot 0. Slot 0 is filled (J4). J3 cannot be scheduled.
4. J2 (10, 1): Try slot 0. Slot 0 is filled (J4). J2 cannot be scheduled.
    - Maximum Profit: 127.
    - Scheduled Jobs: J4, J1.

4. Travelling Salesman Problem (TSP) using Branch and Bound

- Paradigm: Branch and Bound.

- Concept: TSP is a classic NP-hard problem: given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin[7] city. Branch and Bound explores the state space tree, but it systematically prunes branches that are unlikely to lead to an optimal solution by using lower bounds.
- Algorithm (High-Level):
    1. Represent the problem as a state space tree (each node representing a partial tour).
    2. Calculate a lower bound for each node (partial tour). A common lower bound involves finding the sum of the minimum outgoing edge from each unvisited city plus the cost of the path taken so far.
    3. Maintain an upper_bound (cost of the best tour found so far). Initialize with infinity or a heuristic solution.
    4. Use a priority queue (Min-Heap) to store live nodes, prioritizing nodes with lower bounds (suggesting they are closer to the optimal solution).
    5. Repeatedly extract the node with the lowest lower bound from the priority queue.
    6. Branch: Generate children nodes (extend the partial tour by visiting a new city).
    7. Bound: For each child node:
        - Calculate its lower bound.
        - If the child's lower bound is greater than or equal to the current upper_bound, prune the branch (don't explore it further).
        - If a complete tour is formed (leaf node), update upper_bound if this tour is better.
        - Otherwise, add the child node to the priority queue.
- Application Example: (Requires a cost matrix or graph. The image provided seems to be for Floyd-Warshall/Dijkstra, not directly for TSP matrix. Let's use a generic example format as provided in previous papers for TSP matrices):
    - Graph/Matrix: (e.g., from Summer 2022 Q.3 B)
    - A B C D E
    - A | INF 20 30 10 11
    - B | 15 INF 16 4 2
    - C | 35 8 INF 2 4
    - D | 19 6 18 INF 3
    - E | 4 16 7 16 INF
    - Steps (illustrative, full solution is extensive):

1. Reduce rows/columns: Subtract the minimum element from each row, then each column, to get a reduced cost matrix. The sum of these subtracted values is an initial lower bound.

2. Branching: From a starting node (e.g., A), explore paths (A->B, A->C, etc.).

3. Lower Bound Calculation: For each partial tour, calculate a lower bound by adding the cost of the path taken, plus the sum of minimum edge costs from unvisited cities, plus potential minimum entry costs to unvisited cities.

4. Pruning: If a lower bound exceeds the current best tour found, prune that branch.

    - *Note: A complete step-by-step example for TSP with Branch and Bound is quite long and complex, usually demonstrated in class or detailed textbook examples.*

Lower Priority Topics Notes

1. Binary Search

- Paradigm: Divide and Conquer.
- Concept: An efficient algorithm for finding an item from a *sorted* list of items. It repeatedly divides the search interval in half.[8]
- Algorithm:
    1. Compare the target value with the middle element of the array.
    2. If they are equal, the position is found.
    3. If the target is smaller than the middle element, search in the left half.
    4. If the target is larger, search in the right half.
    5. Repeat[9] until the value is found or the interval is empty.
- Time Complexity: O(logn). This is due to the search space being halved in each step.
- Space Complexity: O(1) for iterative, O(logn) for recursive (due to recursion stack).

2. Max and Min Heap & Insertion

- Heap: A complete binary tree that satisfies the heap property.
- Max Heap: For every node i other than the root, Parent(i) >= i. The largest element is always at the root.
- Min Heap: For every node i other than the root, Parent(i) <= i. The smallest element is always at the root.
- Algorithm to Insert into a Max Heap:
    1. Place the new element at the first available position in the last level (maintaining completeness).
    2. Heapify-Up (or Bubble-Up): Compare the new element with its parent. If the new element is larger than its parent, swap them.
    3. Repeat step 2 until the element is smaller than its parent or it reaches the root.
- Algorithm to Insert into a Min Heap:

1. Place the new element at the first available position in the last level.
2. Heapify-Up (or Bubble-Up): Compare the new element with its parent. If the new element is smaller than its parent, swap them.
3. Repeat step 2 until the element is larger than its parent or it reaches the root.

Remember to practice applying these concepts with numerical examples wherever applicable, as most questions in your exam require application and evaluation!