At a glance DAA

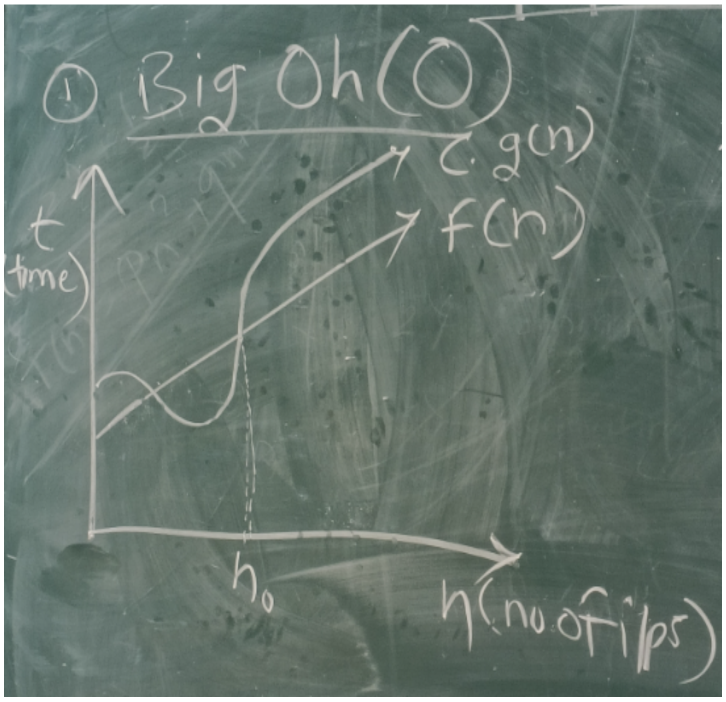
Unit 1

Asymptotic Notation

1. Without execution the time and space complexity of an algorithm is calculated using asymptotic notations
2. Graphs can be drawn where time vs no of inputs can be taken into consideration
3. Types
   a. Big O notation O(n)
      --> By using the Big O notation hard bound can be understood (Upper bound)
      --> it gives the worst-case complexity of an algorithm
   b. Omega notation Ω(n)
      --> Lower bound is determined
      --> it provides the best case complexity of an algorithm
   c. Big theta notation θ(n)
      --> Determine whether our value is in range i.e upper bound and lower bound
      --> it is used for analyzing the average-case complexity of an algorithm

Example O(n)

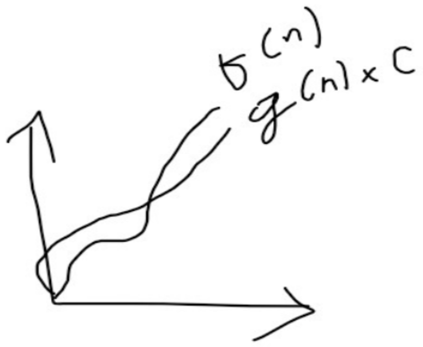
```
f(n) <= c * g(n)

Constraints:

c > 0

n > n0

n0 >= 0

Equations:

    Suppose,
    f(n) = 2n**2 + n
    g(n) = n**2
```



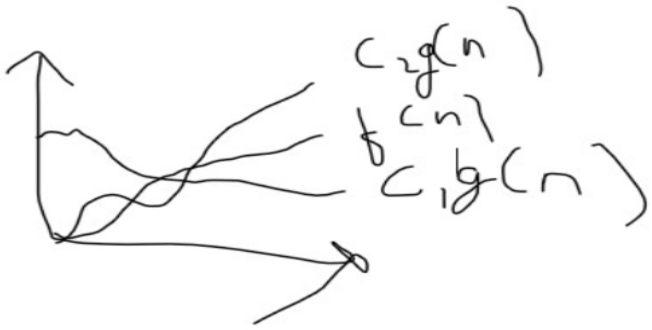
```
    f(n) <= c*g(n)
    2(n**2) + n <= C*(n**2)
    n <= 0
    2(n**2) + n <= 3(n**2)
    n <= (n**2)
    1 <= n
```

Example Ω(n)

```
    Suppose that
    C = 2
    2n*n + n >= c*n*n
    n >= 2n*n - 2n*n
    n >= 0

    Thus lower bound is determined by
Ω(n)
```



Example θ(n)



```
    C1*g(n) < f(n) < C2*g(n)
```

Searching Algorithms

Linear search

i. Best case complexity is 1 (const) $\Omega(1)$
ii. Average case $\theta(n)$
iii. Worst case $O(n)$

Classifying Problems

1. P-Class
    i. Time complexity is O(n**k)
    ii. They are Tractable Problems
    iii. This class contains many natural problems like:
        a. Calculating the greatest common divisor.
        b. Finding a maximum matching.
        c. Decision versions of linear programming.
2. NP-Class (Non-deterministic Polynomial Class)
    i. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify
    ii. Problems of NP can be verified by a Turing machine in polynomial time
    iii. Examples
        a. Boolean Satisfiability Problem (SAT).
        b. Hamiltonian Path Problem.
        c. Graph coloring.
3. Co-NP Class
    i. Co-NP stands for the complement of NP Class
    ii. If a problem X is in NP, then its complement X' is also is in CoNP.
    iii. For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer "yes" or "no" in polynomial time for a problem to be in NP or CoNP.
    iv. Examples
        a.To check prime number
        b. Integer Factorization
4. NP-hard class

    i. All NP-hard problems are not in NP.

ii. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not

iii. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A

iv. Example

a. Halting problem

b. Quantified Boolean formulas

c. No Hamiltonian cycle

5. NP-complete class

i. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time

ii. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time

iii. A problem is NP-complete if it is both NP and NP-hard

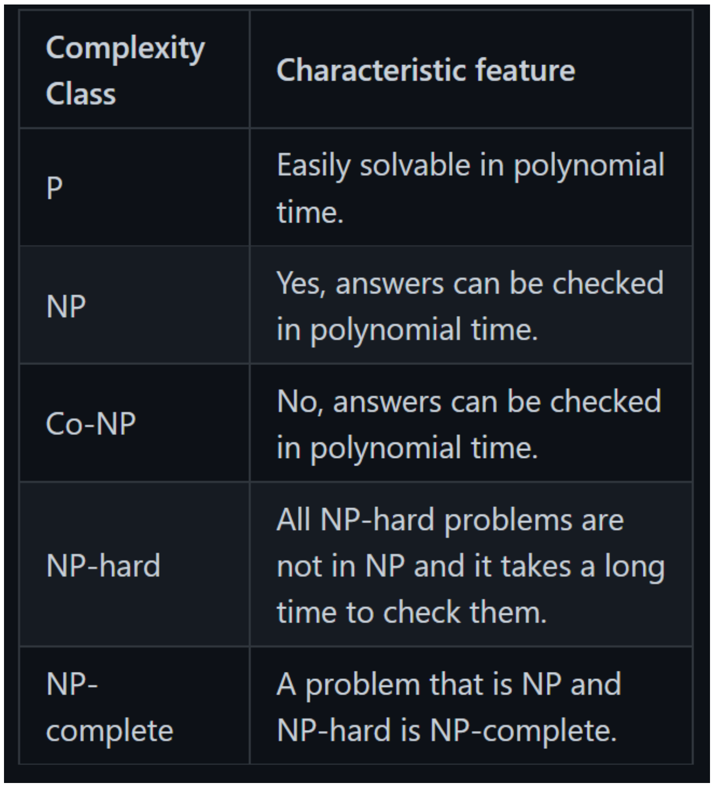
iv. Example

a. Decision version of 0/1 Knapsack

b. Hamiltonian Cycle

c. Satisfiability

d. Vertex cover



Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Problem Solving

```
Important **Steps** to solve problems

    1. Understanding the Problem Statement
    2. Figure Out Sample Inputs and Expected Outputs
    3. Divide the Problem Into Small Chunks
    4. Explore Important Existing Algorithms for Inspiration
    5. Write Pseudocode First
    6. Write Simple Code and Generalize It
    7. Try to Improve Your Algorithmic Creation
    8. Find a Suitable Data Structure
    9. Problem Solving via Design Patterns
    10. Practice Make a Developer Perfect
```

Classification of Time Complexities

```
1. O(1) - Constant Time Complexity

        It does not matter, how many input elements a problem have, it
takes constant number of steps to solve the problem

2. O(log n) - Logarithmic Time complexity

        In every step, halves the input size in the logarithmic algorithm,
log2 n is equal to the number of times n must be divided by 2 to get 1.

        Let us take an array with 16 elements input size, that is - log2
16

        step 1: 16/2 = 8 will become input size

        step 2: 8/2 = 4 will become input size

        step 3: 4/2 =2 will become input size

        step 4: 2/2 =1 calculation completed.
```

The calculation will be performed till we get a value 1.

3. O(n) = Linear Complexity

The number of steps and time required to solve a problem is based on input size. If you want to print a statement for n number of times, it is a linear complexity. Linear Search is an example for Linear Time Complexity

4. O(n log n) - (n * log n) complexity

Divide and Conquer algorithms are generally considered under n log n time complexity.

5. O(n^2) – Quadratic time complexity

If we use a nested loop, that means a loop within another loop, is a quadratic complexity.

Outer loop runs n number of times, inner loop runs n*n number of times, that is O(n2).

6. O(n^n) – A n complexity

In the quadratic algorithm, we used two nested loops. In the cubic algorithm we use n nested loops.

7. O(2^n) – Subset of input elements.

The algorithm must run through all possible subsets of given input.

Ex:

{A, B, C} – possibilities of subsets are 2^3

{ }, {A}, {B}, {C}, {A,B}, {A, C}, {B,C}, {A,B,C}

8. O(n!) – Factorial complexity

```
        Example: All permutations of input elements.
```

Divide and conquer

Concepts are broken into 2 parts from each problem related to the problem statement itself.

```
    Suppose

    n = 2^k

    In first iteration we divide it as n = 2^(n-1)

    In second iteration we divide it as n = 2^(n-2)

    In third iteration we divide it as n = 2^(n-3)

    Then for combining we also need time of

    F(n) = (2^3) * (2^(n-3)) + k (time required for combining the problem)

Simple problem

    T(n) = T(n/2) + T(n/2) + n

    T(n) = 2 * T(n/2) + n

    also T(n/2) = 2 * T(n/4) + n/2

    by Substitution

    T(n) = 2[2*T(n/2) + n/2 ] + n

    T(n) = 4T(n/4) + 2n.

we can expand it so on

    Finally it would look like T(n) = nT(1) + kn
    T(n) = k(n)
```

Why divide and conquer?

1. Easy to use
2. Solves problem in ease

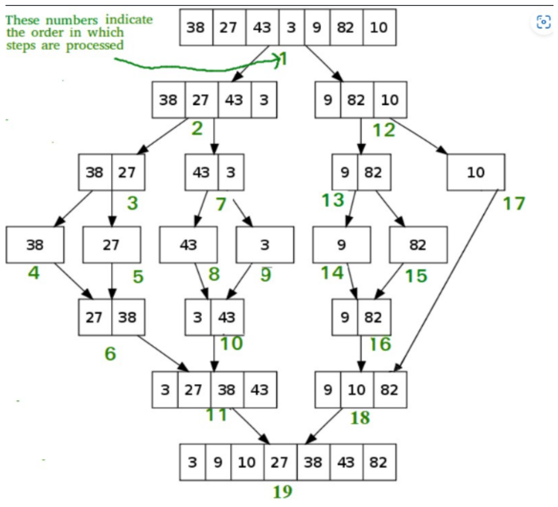
Analysis of merge sort

A merge sort consists of several passes over the input.

1. The first pass merges segments of size 1, the second merges segments of size 2, and the i th pass merges segments of size 2i-1.
2. Thus, the total number of passes is [log2n]. As merge showed, we can merge two sorted segments in linear time, which means that each pass takes O(n) time. Since there are [log2n] passes, the total computing time is O(nlogn).

Merge Sort Working Process

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one



Examples

```
1. State whether the following function is correct or incorrect.


       a. 3n + 2 = O(n) (Remember the graph of O(n))
          Solution a:
               As n >= n0
               f(n) = Og(n)
```

```
                f(n) <= C * g(n)
                f(n) = 3n + 2
                g(n) = n
                by putting n = 1 f(n) will be 5
                so g(n) = C * n
                as n >= n0 so C should be 5
                as n's value it can be >= 1 in order for the statement
f(n) <= C * g(n) to be true.
                Similarly we can also check for n = 2, f(n) will be 8 and
g(n) will be 10 which is exactly our condition

        b. 100n + 6 = O(n)
           Solution b:
                As n >= n0
                f(n) = Og(n)
                f(n) <= C * g(n)
                f(n) = 100n + 6
                g(n) = n
                putting n = 1
                f(n) = 106
                so C = 106
                Checking for n = 2
                f(n) = 206 and g(n) = 212
                so C = 106 and n >= 1 so that will satisfy our condition
f(n) <= C * g(n)

        c. 10(n^2) + 4n + 2 = O(n^2)
           Solution c:
                As n >= n0
                f(n^2) = Og(n^2)
                f(n^2) <= C * g(n^2)
                f(n^2) = 10(n^2) + 4n + 2
                g(n^2) = n^2
                lets say n = 1 in f(n^2)
                f(1) = 10 + 4 + 2 = 16
                so C = 16
                Checking for n = 2
                f(4) = 40 + 16 + 2 = 58
                g(4) = 16 * 4 = 64
```

```
                  so C = 16 and n >= 1 will satisfy our condition f(n^2) <=
C * g(n^2)
```

Master's Theorem

```
Remember the equation T(n) = aT(n/b) + f(n)

where a >= 1 & b > 1

Solution is T(n) = n^(log a to the base b) * U(n)

U(n) depends on h(n)

where h(n) = f(n)/ (n^log a to the base b)

Relation between U(n) & h(n)

if h(n) then U(n)

  n^r, r > 0 then U(n) =  O(n^r)

  n^r, r < 0 then U(n) = O(1)

  (log n)^i, i >= 0 then U(n) = ((log n)^(i+1))/(i+1)

Note: r is the power of h(n) function4

Once U(n) is found then substitute it in T(n) formula
```

Example

```
T(n) = 8 T (n/2) + n log n

a >= 1 & b > 1
```

```
    8 >= 1 & 2 > 1

    so we can use master's theorem here

    T(n) = n^(log 8 to the base 2) + U(n)

    T(n) = n^3 + U(n)

    so h(n) = f(n)/n^(log 8 to the base 2)
    h(n) = nlog(n) / n^3

    h(n) = (n^(-2)) * log(n)

    as we see -2 < 0

    U(n) = 1

    therefore T(n) = n^3 * 1

    T(n) = n^3
```

Quicksort & Mergesort difference

1. QuickSort uses a pivot element to sort, while Merge Sort does not use the pivot element to sort an array.
2. QuickSort is an internal sorting method where the data is sorted in main memory, while Merge Sort is an external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needs auxiliary memory for sorting.
3. Merge Sort is stable as two elements with equal value appear in the same order in the sorted output as they were in the input, while QuickSort is unstable.
4. QuickSort is mostly preferred for large unsorted arrays, while Merge Sort is mostly applicable for linked lists.
5. QuickSort is more efficient and works faster in smaller size arrays, as compared to Merge Sort. Merge Sort is more efficient

```
and works faster in larger data sets or arrays, as compared to
Quick Sort
```

UNIT 2

Greedy method

1. An Optimization problem is one in which the aim is to either maximize or minimize a given objective function w. r. t. some constraints or conditions, given a set of input values
2. Greedy algorithm always makes the choice (greedy criteria) that looks best at the moment, to optimize a given objective function.
3. It makes a locally optimal choice
4. The greedy algorithm does not always guarantee the optimal solution but it generally produces solutions that are very close in value to the optimal

Dynamic Programming

1. Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again

2. There are two main properties of a problem

a. Overlapping Subproblems

In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again

b. Optimal Substructure

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems

a. Memoization (Top Down)

i. The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions

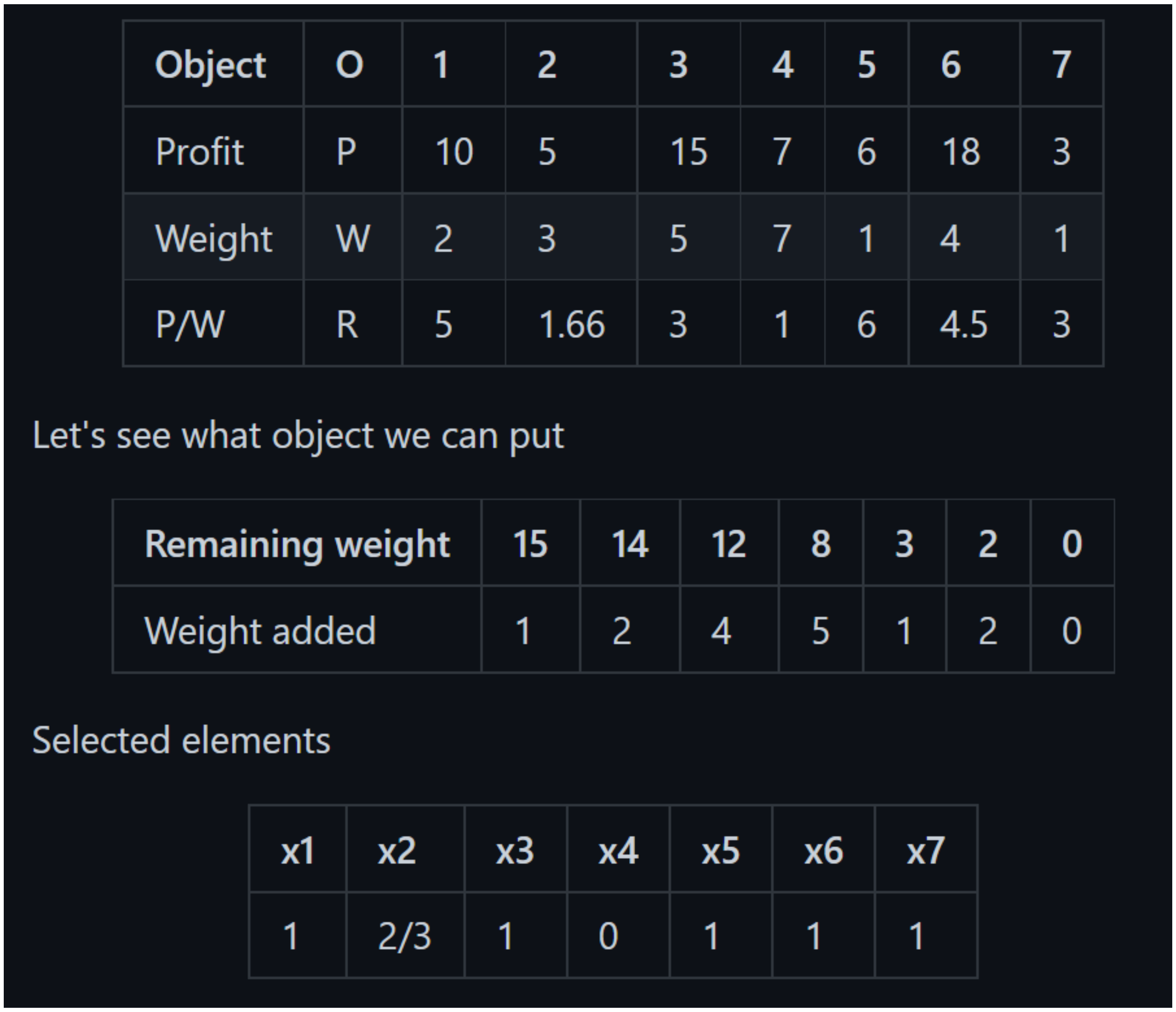
ii. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later

b. Tabulation (Bottom Up)

i. The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table

Knapsack Problem

```
The knapsack problem is the following problem in combinatorial
optimization:

    Given a set of items, each with a weight and a value, determine
which items to include in the collection so that the total weight is
less than or equal to a given limit and the total value is as large
as possible

Lets see it in a broader view

    Suppose
    n (no of objects) =
    m (max weight) = 15
```



Object	O	1	2	3	4	5	6	7
Profit	P	10	5	15	7	6	18	3
Weight	W	2	3	5	7	1	4	1
P/W	R	5	1.66	3	1	6	4.5	3

Let's see what object we can put

Remaining weight	15	14	12	8	3	2	0
Weight added	1	2	4	5	1	2	0

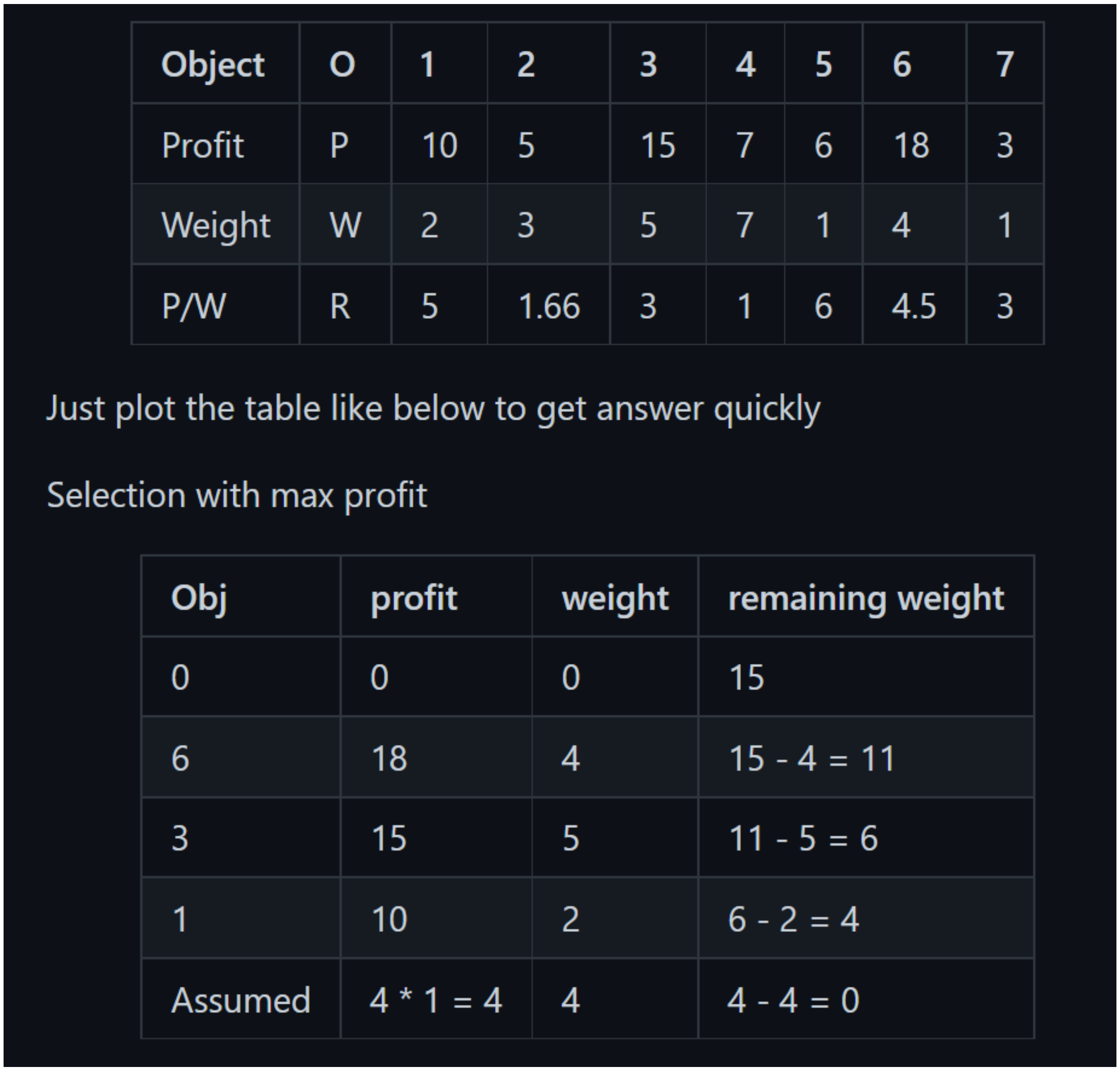
Selected elements

x1	x2	x3	x4	x5	x6	x7
1	2/3	1	0	1	1	1

SUM(xiwi) = 1 x 2 + 2/3 x 3 + 1 x 5 + 1 x 1 + 1 x 4 + 1 x 1 = 15

SUM(xipi) = 1 x 10 + 2/3 x 5 + 1 x 6 + 1 x 18 + 1 x 3 = 55.3

```
Q. Let's see the same problem but with steps again


    n = 7
    m = 5
```



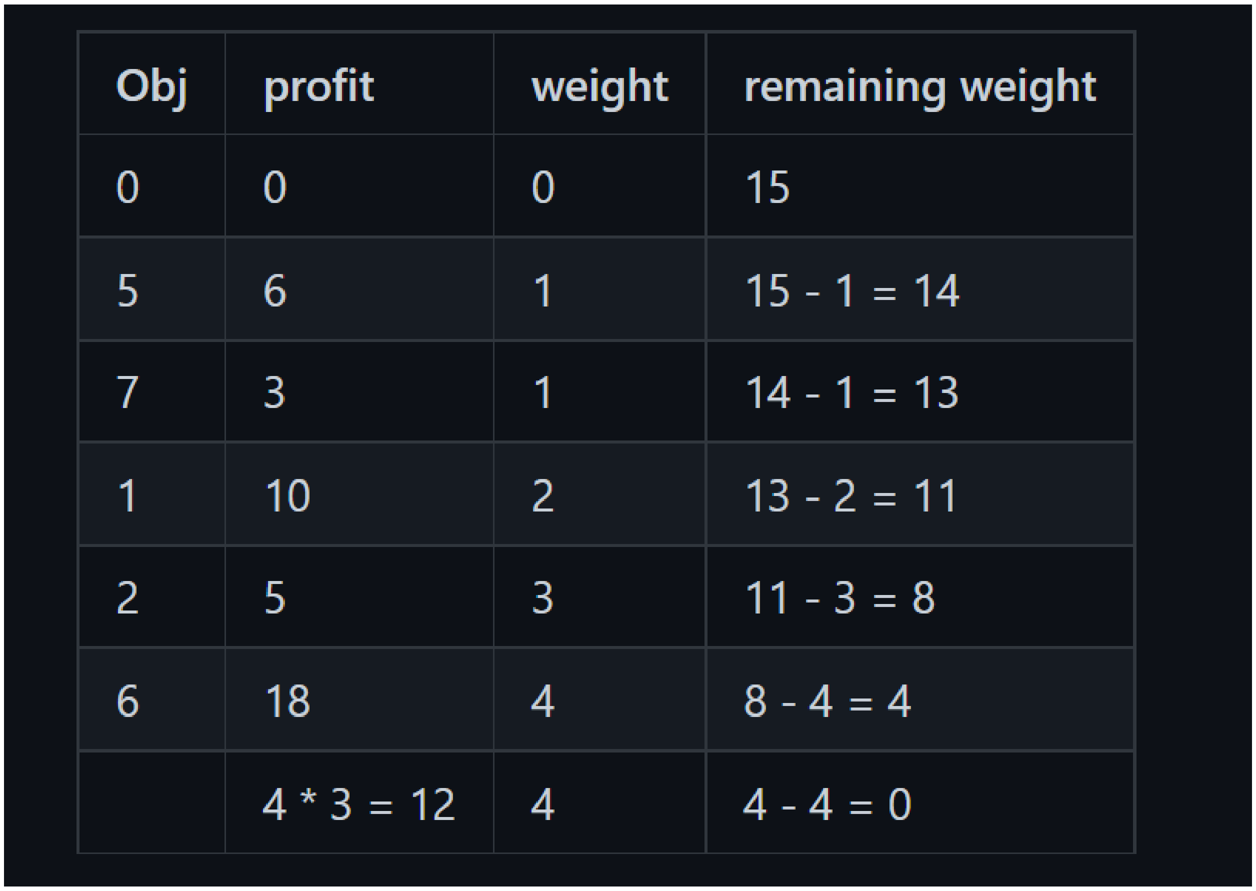
Object	O	1	2	3	4	5	6	7
Profit	P	10	5	15	7	6	18	3
Weight	W	2	3	5	7	1	4	1
P/W	R	5	1.66	3	1	6	4.5	3

Just plot the table like below to get answer quickly

Selection with max profit

Obj	profit	weight	remaining weight
0	0	0	15
6	18	4	15 - 4 = 11
3	15	5	11 - 5 = 6
1	10	2	6 - 2 = 4
Assumed	4 * 1 = 4	4	4 - 4 = 0

```
max profit = 47


Selection with min weight
```



Obj	profit	weight	remaining weight
0	0	0	15
5	6	1	15 - 1 = 14
7	3	1	14 - 1 = 13
1	10	2	13 - 2 = 11
2	5	3	11 - 3 = 8
6	18	4	8 - 4 = 4
	4 * 3 = 12	4	4 - 4 = 0

```
max profit = 54


Select  object with maximum pi/wi
```

Obj	profit	weight	remaining weight
0	0	0	15
5	6	1	15 - 1 = 14
1	10	2	14 - 2 = 12
6	18	4	12 - 4 = 8

3	15	5	8 - 5 = 3
7	3	1	3 - 1 = 2
2	2 * 1.66	2	2 - 2 = 0

```
max profit = 55.34

Because of max pi/wi we will get maximum value
```

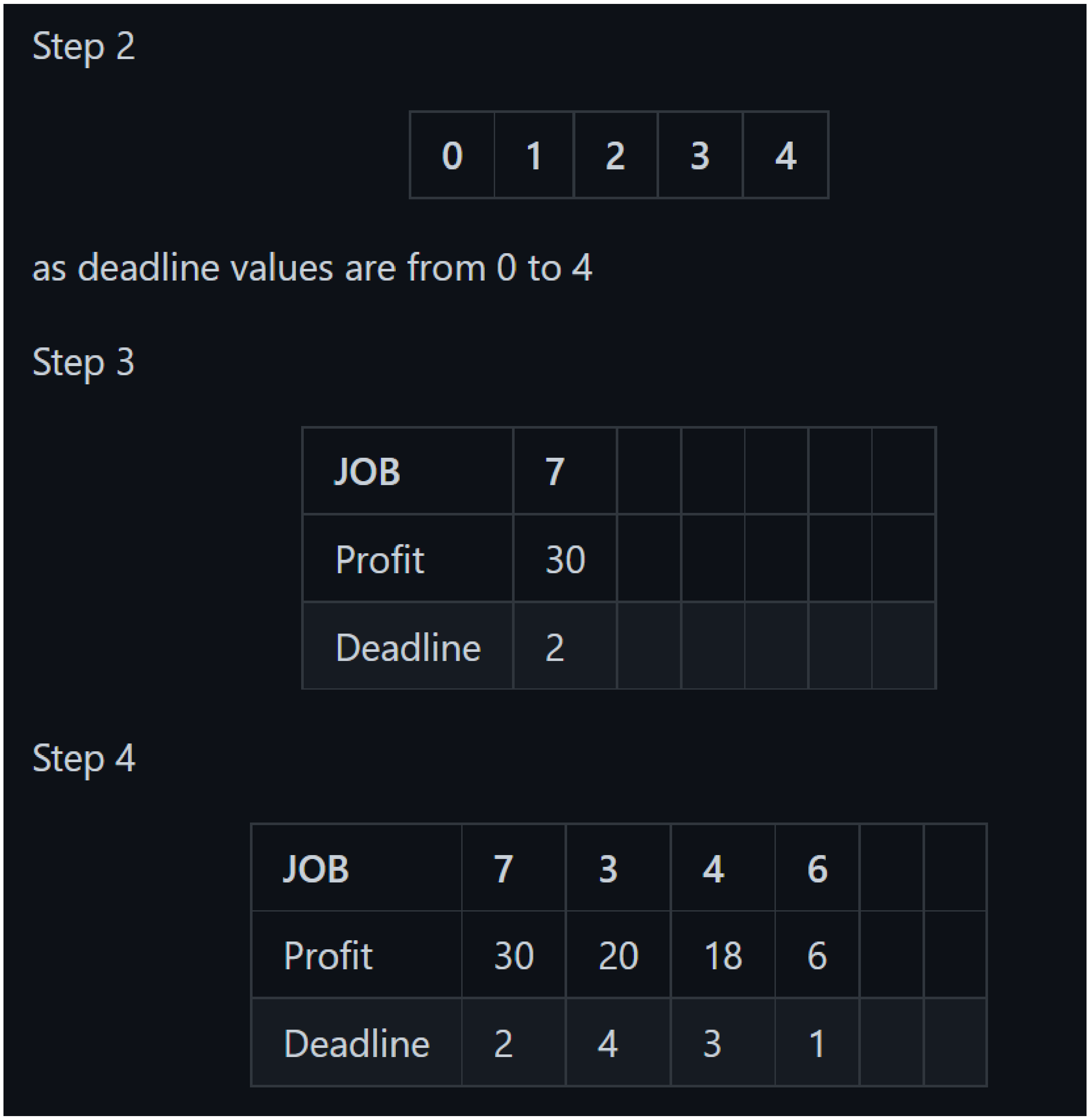
Job Sequencing with deadlines

Given Table

Job	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Profit	3	5	20	18	0	6	30
Deadline	1	3	4	3	2	1	2

Step 1

Job	**7**	**3**	**4**	**6**	**2**	**1**	**5**
Profit	30	20	18	6	5	3	0
Deadline	2	4	3	1	3	1	2



Step 2

0	1	2	3	4

as deadline values are from 0 to 4

Step 3

JOB	7					
Profit	30					
Deadline	2					

Step 4

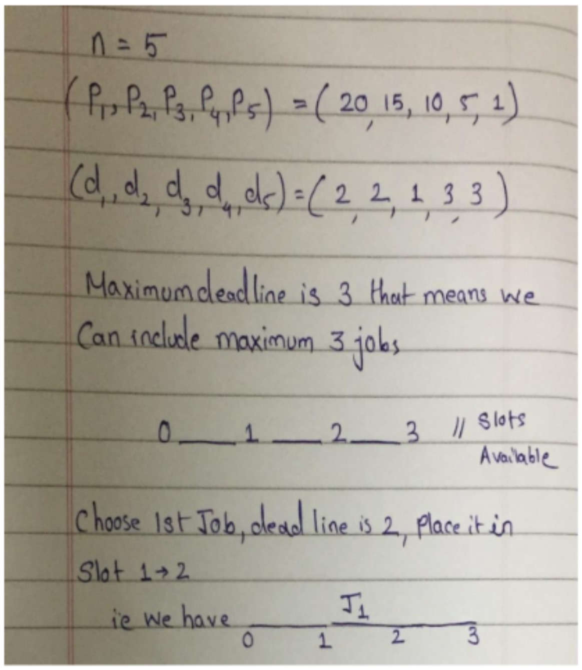
JOB	7	3	4	6		
Profit	30	20	18	6		
Deadline	2	4	3	1		

Step 5

Job	6	7	4	3
Profit	6	30	18	20
Deadline	1	2	3	4

Max profit is 74

Example 2



n = 5

$(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$

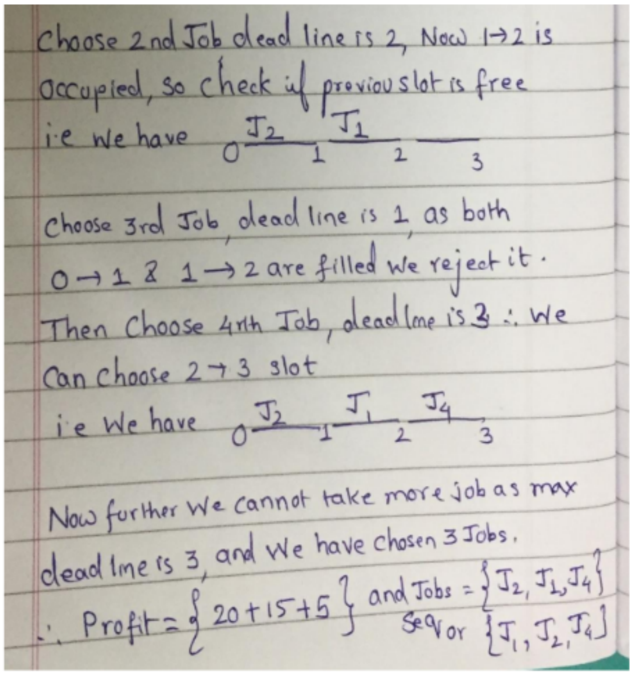
$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

Maximum deadline is 3 that means we Can include maximum 3 jobs

$0 \_\_\_ 1 \_\_\_ 2 \_\_\_ 3$ // Slots Available

Choose 1st Job, dead line is 2, Place it in Slot $1 \to 2$

i.e we have $0 \_\_\_ 1 \underset{}{\overset{J_1}{\_\_\_}} 2 \_\_\_ 3$



Choose 2nd Job dead line is 2, Now $1 \to 2$ is occupied, so check if previous slot is free

i.e we have $0 \overset{J_2}{\_\_\_} 1 \overset{J_1}{\_\_\_} 2 \_\_\_ 3$

Choose 3rd Job, dead line is 1, as both $0 \to 1$ & $1 \to 2$ are filled we reject it.

Then Choose 4rth Job, dead line is 3 $\therefore$ We Can choose $2 \to 3$ slot

i.e We have $0 \overset{J_2}{\_\_\_} 1 \overset{J_1}{\_\_\_} 2 \overset{J_4}{\_\_\_} 3$

Now further We cannot take more job as max dead line is 3, and We have chosen 3 Jobs.

$\therefore$ Profit = $\{20 + 15 + 5\}$ and Jobs = $\{J_2, J_1, J_4\}$ Seq or $\{J_1, J_2, J_4\}$

0/1 Knapsack using DP

```
k(i,c) = max[(Pi + k(i - 1, c - wi),k(i - 1, c))]

k(i,c) = 0 [i == 0 || c == 0]

k(i,c) = k(i,c) [wi > c]
```

Some *Example*

	01	**02**	**03**
P	10	12	28
W	1	2	4

0 to 6 is capacity below and 0 to 3 are objects

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						

```
k(1,1) = max[(10 + k(1,1)),k(1,0)]
k(1,1) = max[10,0]
k(1,1) = 10

k(1,2) = max[(10 + k(1,2)),k(1,2-1)]
k(1,2) = max[10,0]
k(1,2) = 10
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10					
2	0						
3	0						

```
k(1,3) = max[(10 + k(0,2)),k(0,3)]
k(1,3) = max[10,0]
k(1,3) = 10
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0

1	0	10	10	10			
2	0						
3	0						

```
k(1,3) = max[(10 + k(0,2)),k(0,3)]
k(1,3) = max[10,0]
k(1,3) = 10

K(1,4) = K(1,5) = k(1,6) = max(10,0) = 10
```

O/C	**00**	**01**	**02**	**03**	**04**	**05**	**06**
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0						
3	0						

```
K(2,1) = K(2-1,1) = 10
```

O/C	**00**	**01**	**02**	**03**	**04**	**05**	**06**
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10					
3	0						

```
K(2,2) = MAX[12 + K(1,0),K(1,2)]
```

```
K(2,2) = MAX[12,10]
K(2,2) = 12
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12				
3	0						

```
K(2,3) = MAX(12+K(1,1),K(1,3))
K(2,3) = MAX(12+10,10)
K(2,3) = 22
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22			
3	0						

```
K(2,4) = MAX(12+K(1,2),K(1,4))
K(2,4) = MAX(12+10,10)
K(2,4) = 22
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22		

3	0						

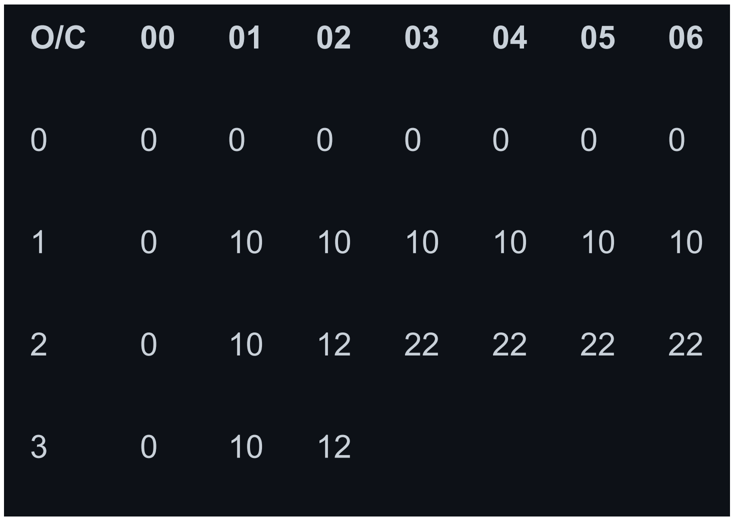
```
K(2,5) = MAX(12+K(1,3),K(1,5))
K(2,5) = MAX(12+10,10)
K(2,5) = 22
K(2,6) = 22
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0						

```
K(3,1) = MAX(10,0)
K(3,1) = 10
```

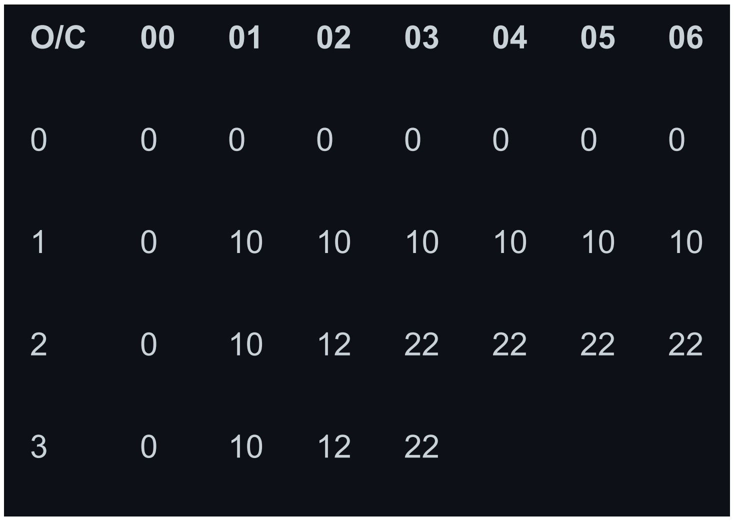
O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10					

```
K(3,2) = 12
```



O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12				

```
K(3,3) = 22
```



O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22			

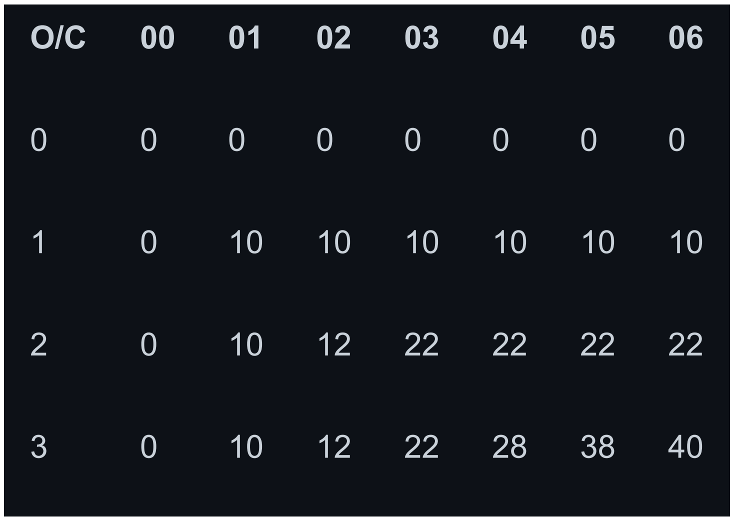
```
K(3,4) = MAX(28 + K(2,0),K(2,4))
K(3,4) = MAX(38,22)
K(3,4) = 28
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28		

```
K(3,5) = MAX(28 + K(2,1),K(2,5))
K(3,4) = MAX(28 + 10 , 22)
K(3,4) = 38
```

O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	

```
K(3,6) = MAX(28 + K(2,2),K(2,6))
K(3,6) = MAX(28 + 12, 22)
K(3,6) = 40
```



O/C	00	01	02	03	04	05	06
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

Max value is 40

Analysis of Knapsack Algo

1. To sort the objects in non-increasing order of pi/wi,computation time = O(nlogn)
2. While loop will be executed 'n' times in worst case, so the computation time = O(n)
3. Time complexity = O(nlogn)
4. Space complexity = c + space required for x[1:n] = O(n)

OBST (An Optimal Binary Search Tree)

- An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost

- In a binary search tree, the search cost is the number of comparisons required to search for a given key

- In an OBST, each node is assigned a weight that represents the probability of the key being searched for being present in the subtree rooted at that node. The tree is constructed such that the expected search cost is minimized

- The optimal binary search tree is generally divided into two types:

1. Static
    a. In the static optimality problem, the tree cannot be modified after it has been constructed
    b. In this case, there exists some particular layout of the tree that minimizes the expected search cost

2. Dynamic
    a. In the dynamic optimality problem, the tree can be modified after it has been constructed
    b. In this case, the tree is constructed incrementally, and the goal is to minimize the expected search cost at each step

IMP Note: Cost contribution of internal node ai is p(i) * level(ai)

How to construct OBST?

- To apply DP approach for obtaining OBST, we need to view the construction of such a tree as the result of sequence of decisions and observe that the principle of optimality holds.
- A possible approach to this would be to decide which of the ai's (with weights pi's) should be selected as the root node of the tree.
- If we choose ak as the root node from a1, a2, …, an (sorted in non decreasing order) , then it is clear that the internal nodes a1, a2, …, ak-1 and external nodes for classes E0, E1, E2, …, Ek-1 will be in the left sub tree l and the internal nodes ak+1, ak+2, …, an and external nodes for classes Ek+1, Ek+2, …, En will be in the right sub tree r. Let root ak be at level 1

Note:

Probability of unsuccessful searches = Σ q[i]

Probability of successful searches = Σ p[i]

Σ p[i] + Σ q[i] = 1

Huffman Coding

Suppose certain string cost 180 bits to send but with the help of huffman coding we can encode it to less than 180 bits

Consider the following text:

ABBCDBCCDAABBEEEBEAB – 20 characters
Now we know that the characters (A B C D E) are repeated in the message.

If there are 4 characters we can represent them with fixed 2 bits as 2 2 = 4 codes.
But we have 5 different characters , thus we will require 3 bit code to represent them as 2 3 = 8.

Thus we can have code as below:
A - 000 B – 001 C – 010 D – 011 E – 100
So now how many bits will be required 20 * 3 = 60 bits

But now along with this 60 bits of encoding the sender will also have to send characters and their codes for receiver to decode the message.

Therefore 8 bits ASCII code for 5 characters = 40
+ Total No. of bits required for encoding (5 * 3) =15

Total = 40 + 15 = 55 bits

Hence the total number of bits that will be sent is 60 + 55 = 115 bits

Multistage graphs

* It's like dijikstra's algorithm

* Finding a minimum cost path from suppose ‘s’ to ‘t’

Solution by DP

Forward Approach :

- cost (i, j) = min{c(j, l) + cost(i+1, l)}

l Є Vi+1 and < j, l > Є E

Backward Approach :

- bcost (i, j) = min{c(l, j) + bcost(i-1, l)}

l Є Vi-1 and < l, j > Є E

Example : Find shortest path from s to t for the above multi-stage graph using a forward and backward approach.
Analysis (Fgraph and Bgraph):

Time Complexity = θ(|V| + |E|)
Space Complexity = θ(n+k) … k is the number of stages.

Unit 3

Branch and Bound

- First we need to conceive a state-space tree for the given problem
- Then generate nodes such that E-node remains E-node till all it's children are generated
- Each answer node x has cost c(x) associated with it, and a minimum

3 common strategies for implementing the branch and bound

a. **FIFO (First In, First Out):**

i.In this strategy, the subproblems are stored in a queue, and the first subproblem to be added to the queue is the first one to be removed for processing.

ii.This strategy is also known as breadth-first search because it explores all the subproblems at a given level of the search tree before moving on to the next level.

b. **LIFO (Last In, First Out):**

i.In this strategy, the subproblems are stored in a stack, and the last subproblem to be added to the stack is the first one to be removed for processing.

ii.This strategy is also known as depth-first search because it explores the deepest branches of the search tree first before backtracking to shallower levels

c. **LC (Least Cost):**

i.In this strategy, the subproblems are sorted by their estimated cost, and the subproblem with the lowest estimated cost is selected for processing first.

ii.This strategy is also known as best-first searchbecause it explores the most promising branches of the search tree first.

iii.The estimated cost can be based on factors such as the distance to the goal, the remaining time, or the number of unsatisfied constraints.

Note:

• FIFO is useful when the solution space is relatively flat and uniform.
• LIFO is useful when the solution space is deep and narrow.
• LC is useful when there is a good heuristic estimate of the optimal solution.

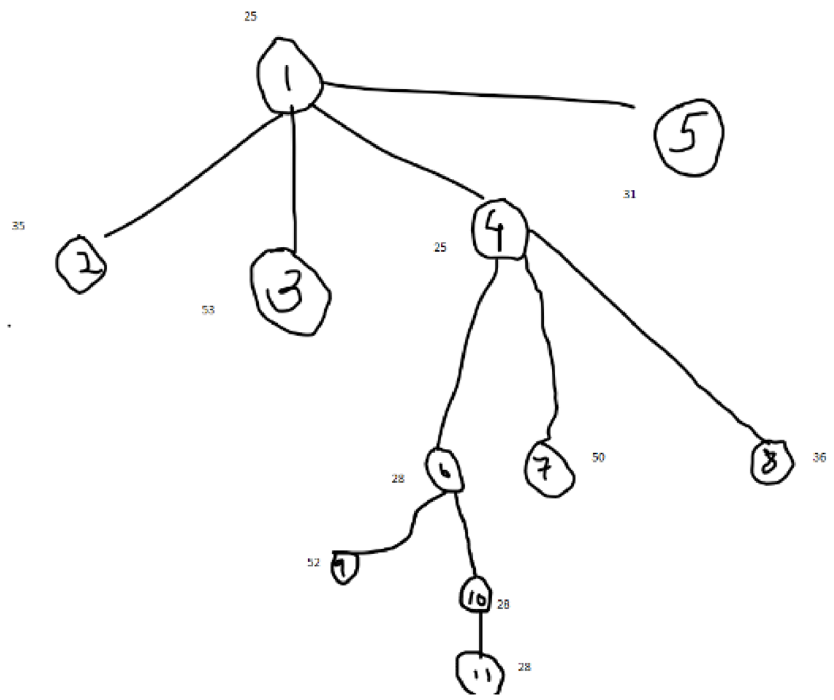
TSP Traveling Salesman Problem

```
    Given a set of cities and distances between every pair of cities,
the problem is to find the shortest possible route that visits every
city exactly once and returns to the starting point.
```

```
M = {

    [INF,20,30,10,11],
    [15,INF,16,4,2],
    [3,5,INF,2,4],
    [19, 6, 18, INF, 3],
    [18,4,7,16,INF]

}

MIN ROW values = [10,2,2,3,4]

reducing the matrix

M = {

    [INF,10 ,20 ,0  ,1  ],
    [13 ,INF,14 ,2  ,0  ],
    [1  ,3  ,INF,0  ,2  ],
    [16 ,3  ,15 ,INF,0  ],
    [13 ,0  ,3  ,12 ,INF]

}

MIN Column values = [1,0,3,0,0]

M = {

    [INF,10 ,17 ,0  ,  1],
    [12 ,INF,11 ,2  ,  0],
    [0  , 3 ,INF,0  ,  2],
    [15 , 3 ,12 ,INF,  0],
    [11 , 0 , 0 ,12 ,INF]

}

consider path 1-2 make 1st row & 2nd column INF

    that is m[2][1] = INF

M = {

    [INF,INF,INF,INF,INF],
    [INF,INF,11 ,2  ,  0],
    [0  ,INF,INF,0  ,  2],
    [15 ,INF,12 ,INF,  0],
    [11 ,INF, 0 ,12 ,INF]

}

val = 25 + 0 + 10 = 35

consider path 1-3 make 1st row & 2nd column INF

    that is m[3][1] = INF

M = {

    [INF,INF,INF,INF,INF],
    [12 ,INF,INF,2  ,  0],
    [INF, 3 ,INF,0  ,  2],
    [15 , 3 ,INF,INF,  0],
    [11 , 0 ,INF,12 ,INF]

}

val = 25 + 11 + 17 = 53

consider path 1-4 make 1st row & 2nd column INF
```

```
    that is m[4][1] = INF

M = {

    [INF,INF,INF,INF,INF],
    [12 ,INF,11 ,INF,  0],
    [0  , 3 ,INF,INF,  2],
    [INF, 3 ,12 ,INF,  0],
    [11 , 0 , 0 ,INF,INF]

}

val = 25

consider path 1-5 make 1st row & 2nd column INF

    that is m[5][1] = INF

M = {

    [INF,INF,INF,INF,INF],
    [12 ,INF,11 ,2  ,INF],
    [0  , 3 ,INF,0  ,INF],
    [15 , 3 ,12 ,INF,INF],
    [INF, 0 , 0 ,12 ,INF]

}

val = 25 + 2 + 3 + 1 = 31

consider path 1-4-2 as infinity

M = {

    [INF,INF,INF,INF,INF],
    [INF,INF,11 ,INF,  0],
    [0  ,INF,INF,INF,  2],
    [INF,INF,INF,INF,INF],
    [11 ,INF, 0 ,INF,INF]

}

val = 25 + 0 + 3 = 28

consider path 1-4-3 as infinity

M = {

    [INF,INF,INF,INF,INF],
    [12 ,INF,INF,INF,  0],
    [INF,3  ,INF,INF,  2],
    [INF,INF,INF,INF,INF],
    [11 ,0  ,INF,INF,INF]

}

val = 25 + 13 + 12 = 50

consider path 1-4-5 as infinity

M = {

    [INF,INF,INF,INF,INF],
    [12 ,INF,11 ,INF,INF],
    [0  , 3 ,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [INF, 0 , 0 ,INF,INF]

}

val = 25 + 11 + 0 = 36

consider path 1-4-2-3

M = {
    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,  2],
    [INF,INF,INF,INF,INF],
    [ 11,INF,INF,INF,INF]

}

val = 28 + 13 + 11 = 52

consider path 1-4-2-5

M = {

    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [  0,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [INF,INF,  0,INF,INF]

}

val = 28 + 0 + 0 = 28

consider path 1-4-2-5-3

M = {

    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF],
    [INF,INF,INF,INF,INF]

}

val = 28 + 0 + 0 = 28
```

Final tree



Q2. Find reduced matrix

```
M = {

    [INF,11  ,10  ,9   ,   6],
    [8   ,INF,7   ,3   ,   4],
    [8   ,4   ,INF,4   ,   8],
    [11  ,10  ,5   ,INF,   6],
    [6   ,9   ,5   ,5   ,INF],

}

min row val = [6,3,4,5,5]

row reduction

M = {

    [INF,5   ,4   ,3   ,   0],
    [5   ,INF,4   ,0   ,   1],
    [4   ,0   ,INF,0   ,   4],
    [6   ,5   ,0   ,INF,   1],
    [1   ,4   ,0   ,0   ,INF],

}

min col values = [1,0,0,0,0]

col reduction

M = {

    [INF,5   ,4   ,3   ,   0],
    [4   ,INF,4   ,0   ,   1],
    [3   ,0   ,INF,0   ,   4],
    [5   ,5   ,0   ,INF,   1],
    [0   ,4   ,0   ,0   ,INF],

}
```

Backtracking

- **Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”**

- **Build up solution vector, one component at a time**

- **Use modified criterion functions Pi(x1, .. ,xn), sometimes called bounding functions to test whether the vector being formed has any chance of success**

- **If partial vector (x1,..xi) cannot lead to success, then mi+1..mn possible test vectors can be ignored entirely**

- **There are 2 categories of constraints that a backtracking problem should statisfy**

A.Implicit
i. All tuples that satisfy the explicit constraints define a possible solution space for I

B.Explicit

i. Rules that determine which of the tuples in the solution space of I satisfy the criterion function

ii. They describe the way in which the xi must relate to each other

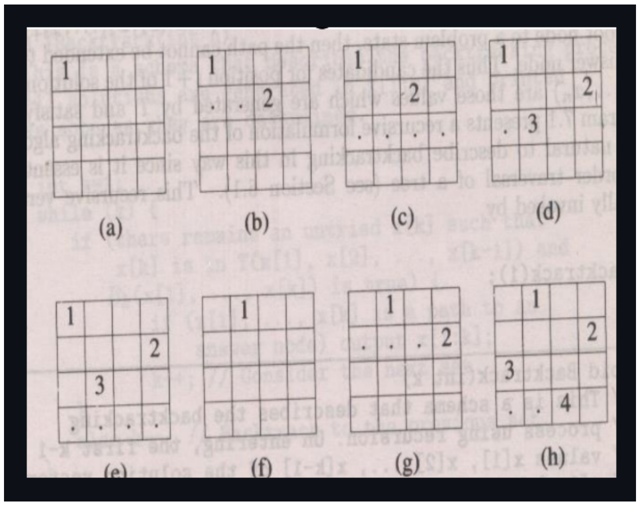
Important Notes:

1. **Backtracking algorithms try each possibility until they find the right one.**
2. **It is a depth-first search of the set of possible solutions.**
3. **During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative**
4. **When the alternatives are exhausted, the search returns to the previous choice point and tries the next alternative there**
5. **If there are no more choice points, the search fails**

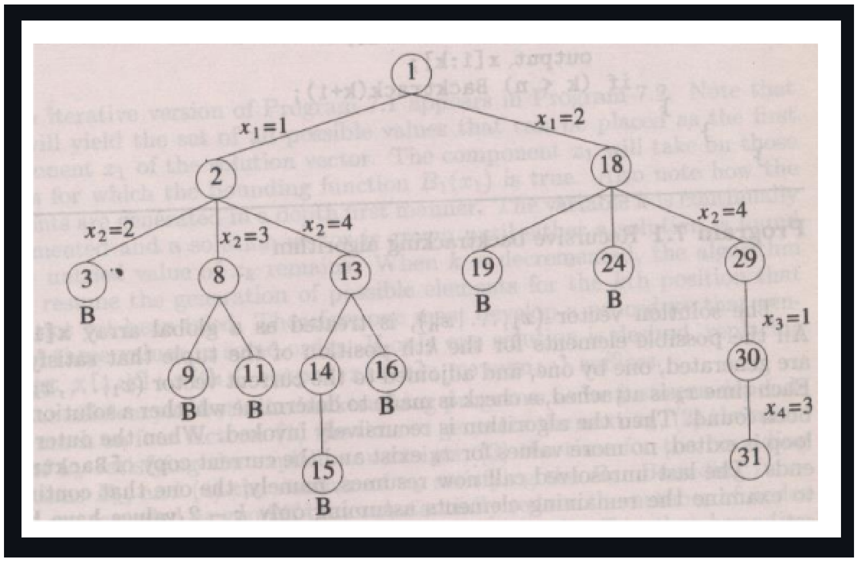
NQueens

- The N-queens problem is a classic problem in computer science
- The problem is to place n queens on an n x n chessboard such that no two queens attack each other
- A queen can attack another queen if it is in the same row, column, or diagonal. The problem can be solved using backtracking
- The time complexity of the N-queens problem is O(n!)
- The space complexity of the N-queens problem is O(n)

Solving NQueen N = 4



Tree generated using backtracking



Graph Coloring Problem

- **Assign colors to vertices in a graph such that no two adjacent vertices have the same color**

Note:

The minimum number of colors required to color a graph such that no two adjacent vertices have the same color is called the chromatic number of the graph.

- **Real life applications:**

  Scheduling problems
  Register allocation in compilers
  Frequency assignment in wireless communication networks etc

- **The problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve the problem exactly for all possible inputs**

- **However, the actual running time of backtracking algorithms can be much better than O(k^n) if the algorithm can prune large portions of the search space using heuristics or other techniques**