ITDO 5014 ADSA SYLLABUS

Module 3 : Divide and Conquer AND Greedy Algorithms : (9 Hrs)

1. Introduction to Divide and Conquer
   Analysis of :
2. Binary search
3. Merge sort and Quick sort
4. Finding the minimum and maximum algorithm

Self-learning Topics:
Implementation of minimum and maximum algorithm, Knapsack problem, Job sequencing using deadlines.

3. Introduction to Greedy Algorithms
4. Knapsack problem
5. Job sequencing with deadlines
6. Optimal storage on tape
7. Optimal merge pattern
8. Analysis of All these algorithms and problem solving.

Contents of Module 3 : Greedy Algorithms :

1. Introduction to Greedy Algorithms
2. Knapsack problem
3. Job sequencing with deadlines
4. Optimal storage on tape
5. Optimal merge pattern
6. Analysis of All these algorithms and problem solving.

Introduction to Greedy Algorithms :

Example :

Problem : Travel from A ---------→ B (250 Km)

Solution S1: by walk **S2: By bike** **S3: Train** **S4: Flight**

Now we add a constraint to this problem that the distance between A and B is to be travelled in 2 Hours.

⇒ **Solutions 1,2 are not feasible.**

⇒ **Solutions 3,4 are feasible.**

The solution which satisfies a given constraint is known as a feasible solution.

Now, we say that the distance should be travelled in minimum cost.

The solution which maximizes the profit or minimizes the cost is a optimal solution.

In general , a problem may have :

multiple solutions (s1, s2,….s4)

multiple feasible solutions (s3, s4)

but only one optimal solution (s3).

Problems that require achieving MIN or MAX result are known as Optimization Problems.

Optimization Problems may be solved using different strategies such as :

1. **Greedy technique**
2. **Dynamic Programming**
3. **Branch and bound**

Greedy is a technique of problem solving and not an Algorithm.

A greedy technique, as the name suggests, always makes the choice that seems to be the best at that moment.

This means that it makes a locally-optimal choice *in the hope* that this choice will lead to a globally-optimal solution.

Thus, Greedy algorithms are used for optimization problems.

An optimization problem can be solved using Greedy if the problem has the following property:

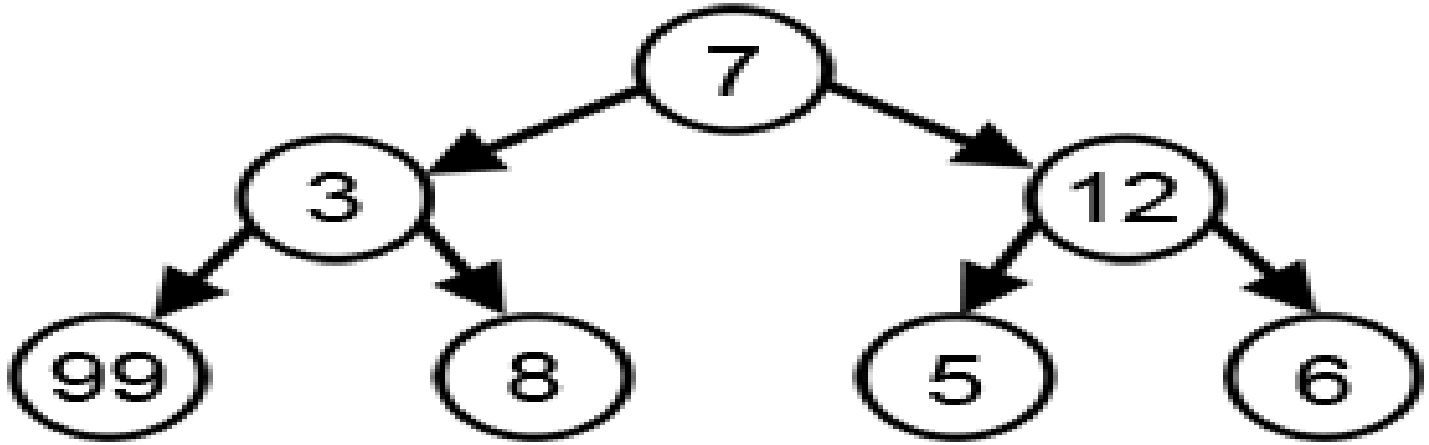
At every step, we can make a choice that looks best at the moment, and
we get the optimal solution of the complete problem.

However, generally greedy algorithms do not provide globally optimized solutions. Eg. Finding Largest Sum (explained in next slide).

Eg1. Finding Largest Sum :

If we've a goal of reaching the largest-sum, at each step,
the greedy algorithm will choose what appears to be the optimal immediate choice,
so it will choose 12 instead of 3 at the second step, and
will not reach the best solution, which contains 99.

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

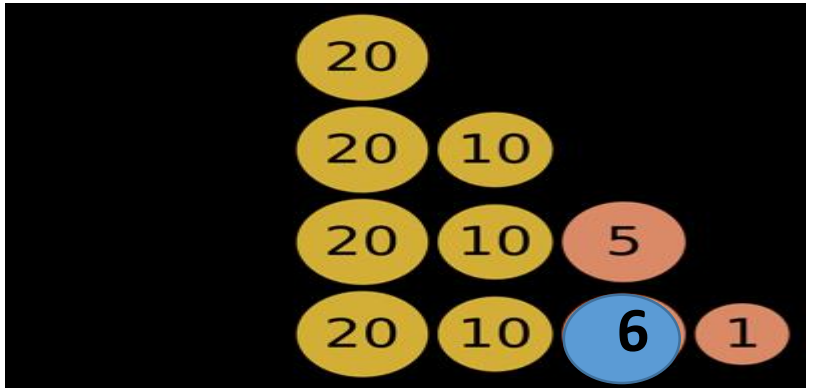


Eg. 2. Counting Coins :

This problem is to count to a desired value by choosing the least possible number of coins .

The greedy approach forces the algorithm to pick the largest possible coin.

If we are provided coins of ₹1, ₹5, ₹10 and ₹20 and we are asked to count ₹36 then the greedy procedure will be –



Select one ₹20 coin, the remaining count is 16
Then select one ₹10 coin, the remaining count is 6
Then select one ₹5 coin, the remaining count is 1
And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins.
But if we slightly change the problem then the same approach may not be able to produce the same optimum result.
For the currency system, where we have coins of ₹1, ₹6, ₹10 and ₹20 value, counting coins for value 36 will be absolutely optimum but for count like 32, it may use more coins than necessary.

For example, the greedy approach will use 20 + 10 + 1 + 1, total 4 coins.
Whereas the same problem could be solved by using only 3 coins (20 + 6 + 6)

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming.

But Greedy algorithms cannot always be applied.

For example, Fractional Knapsack problem can be solved using Greedy, but
0-1 Knapsack cannot be solved using Greedy.

Formal Definition :

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Examples / applications of Greedy Algorithm :

The greedy algorithm is quite powerful and works well for a wide range of problems.

Many algorithms can be viewed as applications of the Greedy algorithms, such as :

- **Travelling Salesman Problem**
- **Prim's Minimal Spanning Tree Algorithm**
- **Kruskal's Minimal Spanning Tree Algorithm**
- **Dijkstra's Minimal Spanning Tree Algorithm**
- **Graph - Map Coloring**
- **Graph - Vertex Cover**
- **Knapsack Problem**
- **Job Sequencing with deadlines Problem**
- **Optimal storage on tapes**
- **Huffman codes (Data Compression)**

Advantages and Disadvantages of Greedy Algorithm :

1. **Finding solution for a problem is quite easy with a greedy algorithm.**

2. **Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer).**

3. **The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues.**

   Even with the correct algorithm, it is hard to prove why it is correct.
   Proving that a greedy algorithm is correct is more of an art than a science.

Contents of Module 4 : Greedy Algorithms :

1. Introduction
2. Knapsack problem
3. Job sequencing with deadlines
4. Optimal storage on tapes
5. Optimal merge pattern
6. Analysis of All problems

Example 1. knapsack problem : Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

objects	Weight	Profit or Value
#1	6	20
#2	5	15
#3	4	10

Knapsack capacity = = W =10

knapsack problem : Two types :

1. Integral (0/1) knapsack : we choose an item entirely (1) or we don't choose it (0).

Thus, we choose object #1 and #3 entirely with a total weight of 10

This gives a total profit value 30. xi = (1, 0, 1)

2. Fractional knapsack : we can choose fractions of items.

The optimal solution is : choose object #1 entirely and 0.8 fraction of object #2.

The total weight is still 10, yet the profit value is now 32.

0.8 * 5 = 4 0.8 * 15 = 12

Thus, allowing fractional solution may give us more valuable solutions.

Fractional Knapsack Problem : Is also known as the continuous knapsack problem.

Q. Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. (10M)

Example : Explain Knapsack Problem. Find an optimal solution to the knapsack instances n = 7, m = 15 and following values for Pi and Wi using Greedy method. (10M)

Profit (Pi) = {10, 5, 15, 7, 6, 18, 3} , Weight (Wi) = {2, 3, 5, 7, 1, 4, 1}
Given: n = 7, m or W = 15, Find an Optimal solution that gives maximum profit.

Step 1: Find profit/ weight ratio.

Step 2: (Arrange this profit/weight ratio in non-increasing order as n values) Since the highest profit/weight ratio is 6. That is p5/w5, so 1st value is 5. Second highest profit/weight ratio is 5. That is p1/w1, so 2nd value is 1. Similarly, calculate such n values and arrange them in non-increasing order. Order = (5, 1, 6, 3, 7, 2, 4)

Step 3 : Select the objects with highest profit by weight to put in the knapsack.

Profit (Pi) = {10, 5, 15, 7, 6, 18, 3} , Weight (Wi) = {2, 3, 5, 7, 1, 4, 1} Given: n = 7, m or W = 15,
Find an Optimal solution using fractional knapsack that gives maximum profit.

Oi	1	2	3	4	5	6	7
Pi	10	5	15	7	6	18	3
Wi	2	3	5	7	1	4	1
Vi = Pi/Wi	10/2 = 5	5/3 = 1.67	15/5 = 3	7/7 = 1	6/1 = 6	18/4 = 4.5	3/1 = 3
Order	2	6	4	7	1	3	5
Xi	1	2/3	1	0	1	1	1

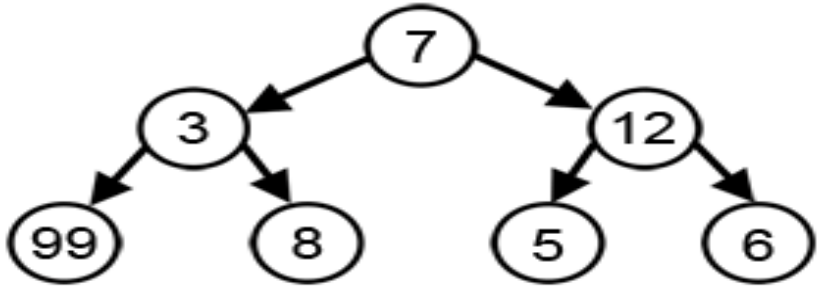
Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack (w)
1	5	6	1	15 – 1 = 14
2	1	5	2	14 - 2 = 12
3	6	4.5	4	12 - 4 = 8
4	3	3	5	8 – 5 = 3
5	7	3	1	3 – 1 = 2
6	2	1.67	3	2 – 2/3 of 3 = 0
7	-	-	-	--

$\sum xiwi = 1*2 + 2/3*3 + 1*5 + 1*1 + 1*4 + 1*1 = 2+2+5+1+4+1 = 15 <= W$
i.e. Capacity of Knapsack
$\sum xiPi = 1*10 + 2/3*5 + 1*15 + 1*6 = 1*18 + 1*3 = 10+3.33+15+6+18+3 = 55.3$
Thus the optimal solution that gives maximum profit =
xi = (1,2/3,1,0,1,1,1)

Q. How do decide whether a particular problem can be solved using Greedy technique ?

Check the problem statement to find if we need to maximize or minimize the goal.
If yes, ➔ This is an optimization problem that can be solved using
Greedy , DP , B&B Technique
Use Greedy : If you desire

1. **Fast solution**
2. **Local optimal solutions /decisions are desirable.**
   Eg. Largest sum at each step – global sum may or may not be Largest.
3. **Can compromise on global optimal solution**



Additional Problems :

Profit (Pi) = {20, 30, 66, 40, 60} , Weight (Wi) = {10, 20, 30, 40, 50}

Given: n = 5, m or W = 100, Find an Optimal solution that gives maximum profit.

Oi	1	2	3	4	5
Pi	20	30	66	40	60
Wi	10	20	30	40	50
Vi = Pi/Wi	20/10 = 2	30/20 = 1.5	66/30 = 2.2	40/40 = 1	60/50 = 1.2
Order	2	3	1	5	4
Xi	1	1	1	0	4/5

Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack (w)
1	3	2.2	30	100 – 30 = 70
2	1	2	10	70 - 10 = 60
3	2	1.5	20	60 - 20 = 40
4	5	1.2	50	40 – 40/50*50 = 0
5	-	-	-	--

∑xiwi = 1*10 + 1*20 + 1*30 + 0 + 40/50*50 = 10+20+30+40 = 100 <= Capacity of Knapsack

∑xiPi = 1*20 + 1*30 + 1*66 + 0*40 + 4/5*60 = 116 + 48 = 164

Thus the optimal solution that gives maximum profit = xi = (1,1,1,0,4/5)

Knapsack Problem:

The Knapsack problem can be stated as follows.

Suppose there are n objects from i=1, 2, …n.
Each object i has some positive weight wi and some profit value Pi is associated with each object .
So its value i.e. price per weight is vi=pi/wi.
The Knapsack can carry at the most weight = W.

Profit (Pi) = {10, 5, 15, 7, 6, 18, 3} , Weight (Wi) = {2, 3, 5, 7, 1, 4, 1}
Given: n = 7, m or W = 15, Find an Optimal solution that gives maximum profit.

Oi	1	2	3	4	5	6	7
Pi	10	5	15	7	6	18	3
Wi	2	3	5	7	1	4	1
Vi = Pi/Wi	10/2 = 5	5/3 = 1.67	15/5 = 3	7/7 = 1	6/1 = 6	18/4 = 4.5	3/1 = 3
Order	2	6	4	7	1	3	5
Xi	1	2/3	1	0	1	1	1

Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack
1	5	6	1	15 – 1 = 14
2	1	5	2	14 - 2 = 12
3	6	4.5	4	12 - 4 = 8
4	3	3	5	8 – 5 = 3
5	7	3	1	3 – 1 = 2
6	2	1.67	3	2 – 2/3 of 3 = 0

∑xiwi = 1*2 + 2/3*3 + 1*5 + 1*1 + 1*4 + 1*1 = 2+2+5+1+4+1 = 15 <= W
i.e. Capacity of Knapsack
∑xiPi = 1*10 + 2/3*5 + 1*15 + 1*6 =1*18 + 1*3 =10+3.33+15+6+18+3 = 55.3
Thus the optimal solution that gives maximum profit =
xi = (1,2/3,1,0,1,1,1)

Aim is to put the objects/items in a knapsack of capacity W to get the maximum total value in the knapsack without exceeding the capacity of Knapsack .

While solving above mentioned Knapsack problem we have the capacity constraint.
When we try to solve this problem using Greedy approach our goal is,

i. Choose only those objects that give maximum profit i.e. MAX ∑ xi * pi .
ii. The total weight of selected objects should be <=W i.e. ∑xi * wi < = W. (capacity constraint)

Thus, we make the greedy choice and pick the most valuable item and take all of it or as much as the knapsack can hold.

Then we move to the second most valuable and do the same.

And then we can obtain the set of feasible solutions,

where the Knapsack can carry the fraction xi of an object i such that 0<=xi<=1 and 1<=i<=n.

This is called the ``fractional knapsack problem'' since we can take a fraction of an item.

Profit (Pi) = {10, 5, 15, 7, 6, 18, 3} , Weight (Wi) = {2, 3, 5, 7, 1, 4, 1}
Given: n = 7, m or W = 15, Find an Optimal solution that gives maximum profit.

Oi	1	2	3	4	5	6	7
Pi	10	5	15	7	6	18	3
Wi	2	3	5	7	1	4	1
Vi = Pi/Wi	10/2 = 5	5/3 = 1.67	15/5 = 3	7/7 = 1	6/1 = 6	18/4 = 4.5	3/1 = 3
Order	2	6	4	7	1	3	5
Xi	1	2/3	1	0	1	1	1

Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack
1	5	6	1	15 – 1 = 14
2	1	5	2	14 - 2 = 12
3	6	4.5	4	12 - 4 = 8
4	3	3	5	8 – 5 = 3
5	7	3	1	3 – 1 = 2
6	2	1.67	3	2 – 2/3 of 3 = 0

∑xiwi = 1*2 + 2/3*3 + 1*5 + 1*1 + 1*4 + 1*1 = 2+2+5+1+4+1 = 15 <= W
i.e. Capacity of Knapsack
∑xiPi = 1*10 + 2/3*5 + 1*15 + 1*6 =1*18 + 1*3 =10+3.33+15+6+18+3 = 55.3
Thus the optimal solution that gives maximum profit =
xi = (1,2/3,1,0,1,1,1)

0/1 Knapsack Problem:

In "0-1 knapsack problem" , we either take all (1) or none (0) of an item.

Profit (Pi) = {10, 5, 15, 7, 6, 18, 3} , Weight (Wi) = {2, 3, 5, 7, 1, 4, 1}
Given: n = 7, m or W = 15, Find an Optimal solution that gives maximum profit.

Oi	1	2	3	4	5	6	7
Pi	10	5	15	7	6	18	3
Wi	2	3	5	7	1	4	1
Vi = Pi/Wi	10/2 = 5	5/3 = 1.67	15/5 = 3	7/7 = 1	6/1 = 6	18/4 = 4.5	3/1 = 3
Order	2	6	4	7	1	3	5
Xi	1	~~2/3~~	1	0	1	1	1

If we are given n objects and a Knapsack or a bag in which the object i that has weight wi is to be placed, The Knapsack has a capacity W.

Then the profit that can be earned is pi*xi.

The objective is to obtain filling of Knapsack with maximum profit earned.

Where 1<=i<=n , n is total number of objects and xi=0 or 1.

In Fractional Knapsack , the Knapsack can carry the fraction xi of an object i such that 0<=xi<=1 and 1<=i<=n.

Algorithm :

algorithm FractionalKnapsack(S,W):

Input: Set S of items i with weight wi and profit / price / benefit pi all positive.
Knapsack capacity W>0.

Output: Amount xi of i that maximizes the total profit / price / benefit without exceeding the capacity of Knapsack .

```
for each i in S do      O(n)
  xi ← 0       { for items not chosen in next phase }
  vi ←pi/wi    { the value of item i "per weight" }
w ← W          { remaining capacity in knapsack }

while w > 0 do    O(n)
  { greedy choice }
  remove from S an item of maximal value
  wi ← min(wi,w)  { can't carry more than w load }
  w ← w-wi
```

W = 100

Oi	1	2	3	4	5
Pi	20	30	66	40	60
Wi	10	20	30	40	50
Pi/Wi	20/10 = 2	30/20 = 1.5	66/30 = 2.2	40/40 = 1	60/50 = 1.2
Order	2	3	1	5	4
Xi	1	1	1	0	4/5

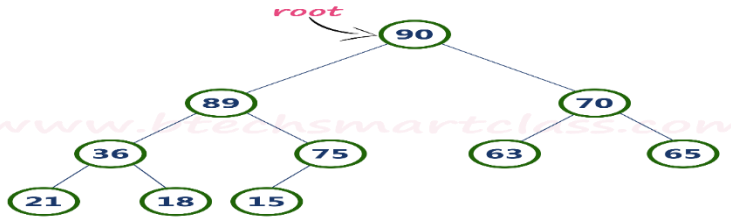
Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack w
1	3	2.2	30	100 – 30 = 70
2	1	2	10	70 - 10 = 60
3	2	1.5	20	60 - 20 = 40
4	5	1.2	50	40 – 40/50*50 = 0

Time complexity :

1. Fractional Knapsack has time complexity O(N logN) where N is the number of items in S.

Assuming S is a heap-based priority queue , the removal has complexity Θ(log N).

so the up to N removals take O(N logN).



The rest of the algorithm is O(N).

So, Time complexity of Fractional Knapsack = O(N log N) + O(N) = O(N log N)

2. Alternatively, S could be a sequence and we could begin Fractional Knapsack by sorting S with a Θ(N logN) sort.

Oi	4	2	5	1	3
Vi = Pi/wi	8	7	4	3	1

Now the removal is simply removing the first element.

If we use a circular list for S, then the removal is O(1) so, N removals take O(N). Thus, the algorithm is O(N).

Including the sort we again have :

Time complexity of Fractional Knapsack = O(N log N) + O(N) = O(N log N)

Why Greedy technique does not work for the normal 0/1 knapsack problem when we must take all of an item or none of it?

Example: knapsack capacity = W=6,
Weights : w1=4, w2=w3=3,
Price : p1=5, p2=p3=3.
Vi = Pi/wi : v1=5/4, v2=v3=1

Oi	1	2	3
pi	5	3	3
wi	4	3	3
Vi = pi/wi	5/4 = 1.25	3/3 = 1	3/3 = 1

Start with the most valuable item i.e. the item with highest Vi value , i.e. object number 1 and put it n the knapsack.

Knapsack can hold 6-4=2 more units of weight but remaining items won't fit
So the total price / benefit carried is 5.

The right solution is to take items 2 and 3 for a total benefit of 6.

The difference between 0/1 and fractional knapsack is that the knapsack can still hold 2/3 of item 2, but we can't take part of an item as we can in the fractional knapsack problem.

Given: n = 4, m or W = 10, Find an Optimal solution that gives maximum profit.

Oi	1	2	3	4
Pi	15	14	12	20
Wi	3	2	2	5
Pi/Wi	15/3 = 5	14/2 = 7	12/2 = 6	20/5 = 4
Order	3	1	2	4
Xi	1	1	1	3/5

Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack
1	2	7	2	10 – 2 = 8
2	3	6	2	8 - 2 = 6
3	1	5	3	6 - 3 = 3
4	4	4	5	3 – 3/5*5 = 0

$\sum xiwi = 1*3 + 1*2 + 1*2 + 3/5*5 =$
$3+2+2+3 = 10 <= \text{Capacity of Knapsack}$

$\sum xiPi = 1*15 + 1*14 + 1*12 + 3/5*20 =$
$= 15 + 14 + 12 + 12 = 43$
Thus the optimal solution that gives maximum profit = xi = (1,1,1,3/5)

How to make an optimal choice ?

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. (Eg. Maximize the profit or minimize the cost)

A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.

The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Item (Oi)	1	2	3	4
Value (Pi)	15	14	12	25
Size (wi)	3	2	2	5

~~Profit (Pi) = {20, 30, 66, 40, 60} , Weight (Wi) = {10, 20, 30, 40, 50}~~

Given: n = 4, m or W = 10, Find an Optimal solution that gives maximum profit.

Oi	1	2	3	4
Pi	15	14	12	20
Wi	3	2	2	5
Pi/Wi	15/3 = 5	14/2 = 7	12/2 = 6	20/5 = 4
Order	3	1	2	4
Xi	1	1	1	3/5

Order	Oi	Pi/wi	Wi	Remaining capacity of Knapsack
1	2	7	2	10 – 2 = 8
2	3	6	2	8 - 2 = 6
3	1	5	3	6 - 3 = 3
4	4	4	5	3 – 3/5*5 = 0

$\sum xiwi = 1*3 + 1*2 + 1*2 + 3/5*5 =$
$3+2+2+3 = 10 <= $ Capacity of Knapsack

$\sum xiPi = 1*15 + 1*14 + 1*12 + 3/5*20 =$
$= 15 + 14 + 12 + 12 = 43$

Thus the optimal solution that gives maximum profit = xi = (1,1,1,3/5)

Contents of Module 3 : Greedy Algorithms :

1. Introduction to Greedy Algorithms
2. Knapsack problem
3. Job sequencing with deadlines
4. Optimal storage on tape
5. Optimal merge pattern
6. Analysis of All these algorithms and problem solving.

3. Job sequencing with deadlines :

Q1. Write a note on Job sequencing with deadlines (10M)

Q2. Give solution using Job sequencing with deadlines for the following instance (10M)

Given an array of jobs { j1, j2, j3, ……..jn } where

every job has a deadline { d1, d2, d3, …….. dn } and

associated profit { p1, p2 , p3 , …… pn } if the job is finished before the deadline.

It is also given that every job takes single unit of time,

How to maximize total profit if only one job can be scheduled at a time ?

Job sequencing Problem with deadlines is :

What is the sequence in which the jobs should be executed so that each job completes before its deadline and together they give maximum total profit .

Algorithm Job sequencing with deadlines :

1) Sort all jobs in decreasing order of profit.

2) Initialize the result sequence as first job in sorted jobs.

3) Do following for remaining n-1 jobs

If the current job can fit in the current result sequence without missing the deadline, add current job to the result.

Else ignore the current job.

Ex1. Explain Job sequencing with deadlines for the following instance (10M)

N=5,

Jobs	J1	J2	J3	J4	J5
Profits	20	15	10	5	1
Deadlines	2	2	1	3	3

Each job needs only one unit of time to perform.

Job sequencing Problem with deadlines is :

What is the sequence in which the jobs should be executed so that each job completes before its deadline and together they give maximum total profit .

0----------1----------2----------3

J2 J1 J4

0----------1----------2----------3

15 20 5

Max Profit Sequence of jobs = {J2, J1, J4}

Max Total Profit = 15 + 20 + 5 = 40

Algorithm Job sequencing with deadlines :

1) Sort all jobs in decreasing order of profit.
2) Initialize the result sequence as first job in sorted jobs.
3) Do following for remaining n-1 jobs

   If the current job can fit in the current result sequenc without missing the deadline, add current job to the result.

   Else ignore the current job.

Ex2. Explain Job sequencing with deadlines for the following instance (10M)

N=5,

Jobs	A	B	C	D	E
Profits	100	19	27	25	15
Deadlies	2	1	2	1	3

Jobs	A	C	D	B	E
Profits	100	27	25	19	15
Deadlies	2	2	1	1	3

Each job needs only one unit of time to perform.

Job sequencing Problem with deadlines is :

What is the sequence in which the jobs should be executed so that each job completes before its deadline and together they give maximum total profit .

0----------1----------2----------3

C A E

0----------1----------2----------3

27 100 15

Max Profit Sequece of jobs = {C, A, E}

Max Total Profit = 27 + 100 + 15 = 132

Algorithm Job sequencing with deadlines :

1) Sort all jobs in decreasing order of profit.
2) Initialize the result sequence as first job in sorted jobs.
3) Do following for remaining n-1 jobs
   If the current job can fit in the current result sequenc without missing the deadline, add current job to the result.
   Else ignore the current job.

Ex3. Explain Job sequencing with deadlines for the following instance (10M)

N=5,

Jobs	A	B	C	D
Profits	20	10	40	30
Deadlies	4	1	1	1

Jobs	C	D	A	B
Profits	40	30	20	10
Deadlies	1	1	4	1

Each job needs only one unit of time to perform.

Job sequencing Problem with deadlines is :

What is the sequence in which the jobs should be executed so that each job completes before its deadline and together they give maximum total profit .

```
0-----1------2------3---------4
      C                       A
0--------1---------2--------3---------4
      40                      20
    C         A
0--------1---------2--------3---------4
    40        20
```

Max Profit Sequece of jobs = {C, A}

Max Total Profit = 40 + 20 = 60 The other jobs B and D can not be executed before their deadline

Algorithm Job sequencing with deadlines :

1) Sort all jobs in decreasing order of profit.
2) Initialize the result sequence as first job in sorted jobs.
3) Do following for remaining n-1 jobs
   If the current job can fit in the current result sequence without missing the deadline, add current job to the result.
   Else ignore the current job.

Algorithm Job sequencing with deadlines :

1) Sort all jobs in decreasing order of profit.

2) Initialize the result sequence as first job in sorted jobs.

3) Do following for remaining n-1 jobs

If the current job can fit in the current result sequence without missing the deadline, add current job to the result.

Else ignore the current job.

```
// Program to find the maximum profit job
// sequence from a  given array of jobs with
// deadlines and profits.
// A structure to represent a job
struct Job
{
  char id;      // Job Id
  int dead;    // Deadline of job
  int profit;  // Profit if job is over before or on
               // deadline
};

// This function is used for sorting all jobs
//  according to profit.
bool comparison(Job a, Job b)
{
   return (a.profit > b.profit);
}
```

```
// Returns Maximum profit  order of jobs

void printJobScheduling(Job arr[], int n)
{
   // Sort all jobs according to decreasing order of profit
  sort(arr, arr+n, comparison);  Time : O(n log n )

   int result[n]; // To store result (Sequence of jobs)
   bool slot[n];  // To keep track of free time slots

   // Initialize all slots to be free
   for (int i=0; i<n; i++)          Time : O(n)
      slot[i] = false;
```

Jobs	A	B	C	D
Profits	20	10	40	30
Deadlies	4	1	1	1

```
// Iterate through all given jobs
  for (int i=0; i<n; i++)          Time : O(n)
  {
    // Find a free slot for this job (Note that we
    // start from the last possible slot)

    for (int j=min(n, arr[i].dead)-1; j>=0; j--)
    {                                  Time : O(n)
      // Free slot found
      if (slot[j]==false)
      {
        result[j] = i;  // Add this job to result
        slot[j] = true; // Make this slot occupied
        break;
      }
    }
  }   Time of above two nested for loops together = O(n^2)
```

```
// Print the result

   for (int i=0; i<n; i++)          Time : O(n)
     if (slot[i])
       cout << arr[result[i]].id << " ";
}

// Driver program to test methods
int main()
{
   Job arr[] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},
               {'d', 1, 25}, {'e', 3, 15}};
   int n = sizeof(arr)/sizeof(arr[0]);
   cout << "Following is maximum profit
   sequence of jobsn";
   printJobScheduling(arr, n);
   return 0;
}
```

Time Complexity of the above solution for Job sequencing with deadlines is :

Sorting array according to decreasing order of profits : O(n log n).
The other sequencing part of the code takes : $n + n^2 + n$ = $O(n^2)$ time .

➔ **Total time = $n^2 + 2n + n \log n = O(n^2)$ time .**

Contents of Module 3 : Greedy Algorithms :

1. Introduction to Greedy Algorithms
2. Knapsack problem
3. Job sequencing with deadlines
4. Optimal storage on tape
5. Optimal merge pattern
6. Analysis of All these algorithms and problem solving.

Optimal Merge Pattern :

Optimal Merge Pattern involves merging a set of sorted files of different lengths into a single sorted file.

We need to find an optimal solution, where the resultant file will be generated in minimum time.

Given, a number of sorted files, there are many ways to merge them into a single sorted file, each way requiring a different amount of time as explained in example.

One way is to merge the files pair wise (i.e. merge two files at a time).
Hence, this type of merging is called as 2-way merge patterns.

To merge two files containing m and n records resp. , the cost of merging them in a single file i.e. the number of comparisons needed to merge them = m+n -1.

Two-way merge patterns can be represented by binary merge trees.

Let us consider a set of n sorted files {f1, f2, f3, …, fn}.

Initially, each element of this is considered as a single node binary tree.

To find this optimal solution, the following algorithm is used.

Algorithm: TREE (n)

```
for i := 1 to n – 1 do      O(n-1)
  declare new node called node
  node.leftchild := least (list)          O(1)      // First smallest ( i.e.  least ) element in the list.
  node.rightchild := least (list)         O(1)     // Second smallest ( i.e.  least ) element in the list.
  node.weight := ((node.leftchild).weight) + ((node.rightchild).weight)
  insert (list, node);                ( log n )      //  O (log n ) , O(1) :If list is maintained as a MIN HEAP
return least (list);                  T(n) = O(n-1) * O (log n) =  O(n) * O( log n ) = O( n log n )
```

The main for loop in this algorithm is executed in n-1 times.

If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only O(1) = constant time and

If the list is represented as a min-heap in which the root value is less than or equal to the values of its children, then
least (list) and insert (list, node) can be done in O (log n) time.

T(n) = O(n-1) * O (log n) = O(n) * O(log n) = O(n log n)

Thus, the computing time for the tree is O (n log n).

To merge two files containing m and n records resp. , the cost of merging them in a single file i.e. the number of comparisons needed to merge them = m+n -1.

Example

Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations is

50 + 60 + 65 + 95 = 270

Is there any better solution?

Sorting the files according to their size in an ascending order, we get the following sequence –

f_4,	f_3,	f_1,	f_2,	f_5
5	10	20	30	30

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

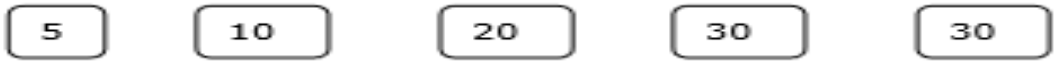
M_4 = merge M_3 and f_5 => 65 + 30 = 95

Therefore, the total number of operations is

15 + 35 + 65 + 95 = 210

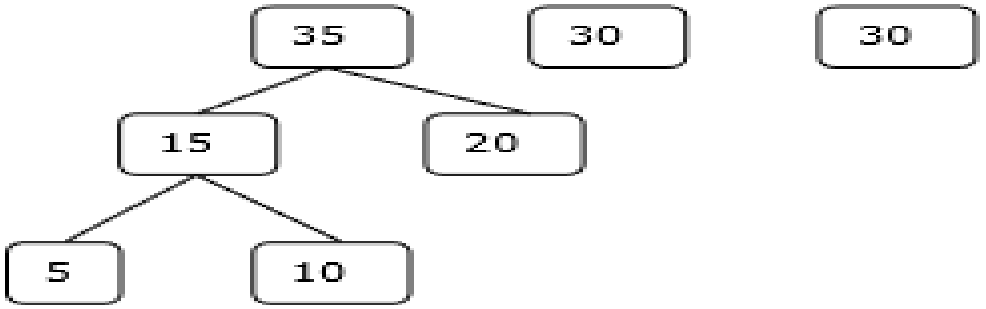
Obviously, this is better than the previous one.

But the optimal Solution is found when we always select a pair of smallest size lists/files to merge as shown below..

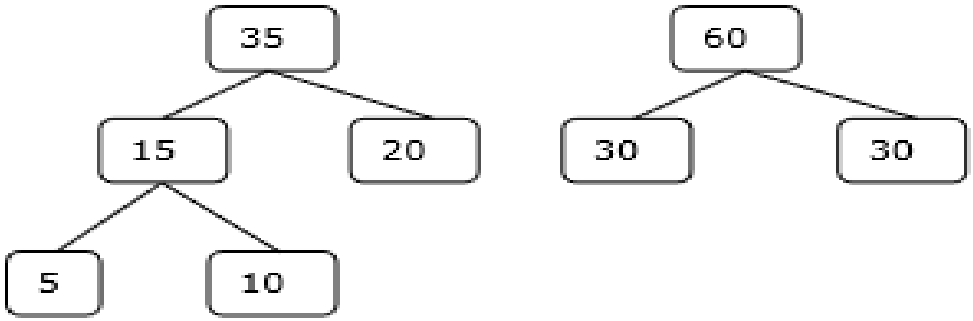


1.

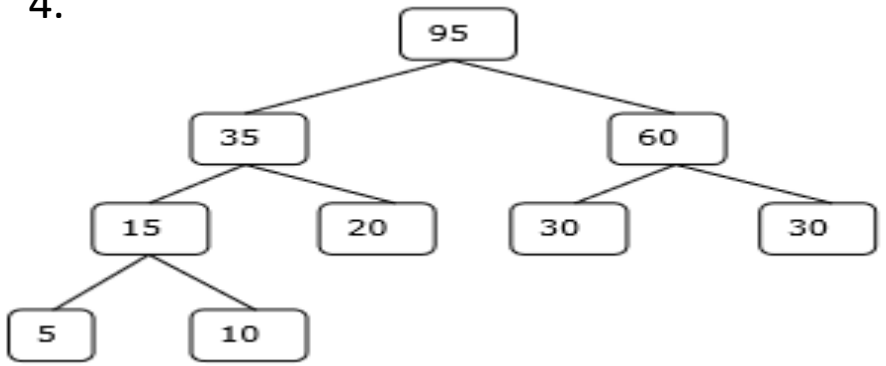
2.



3.



4.



Hence, the solution takes :
15 + 35 + 60 + 95 = 205
number of comparisons.

Ex2 : Merge the files with following lengths using Optimal Merge Pattern.

3, 5, 15 , 7, 17, 9, 14, 12

Step1: Arrage files in increasing order of their length.

3, 5, 7, 9, 12, 14, 15, 17.

Step2: At each step, always select a pair of smallest size lists/files to merge.

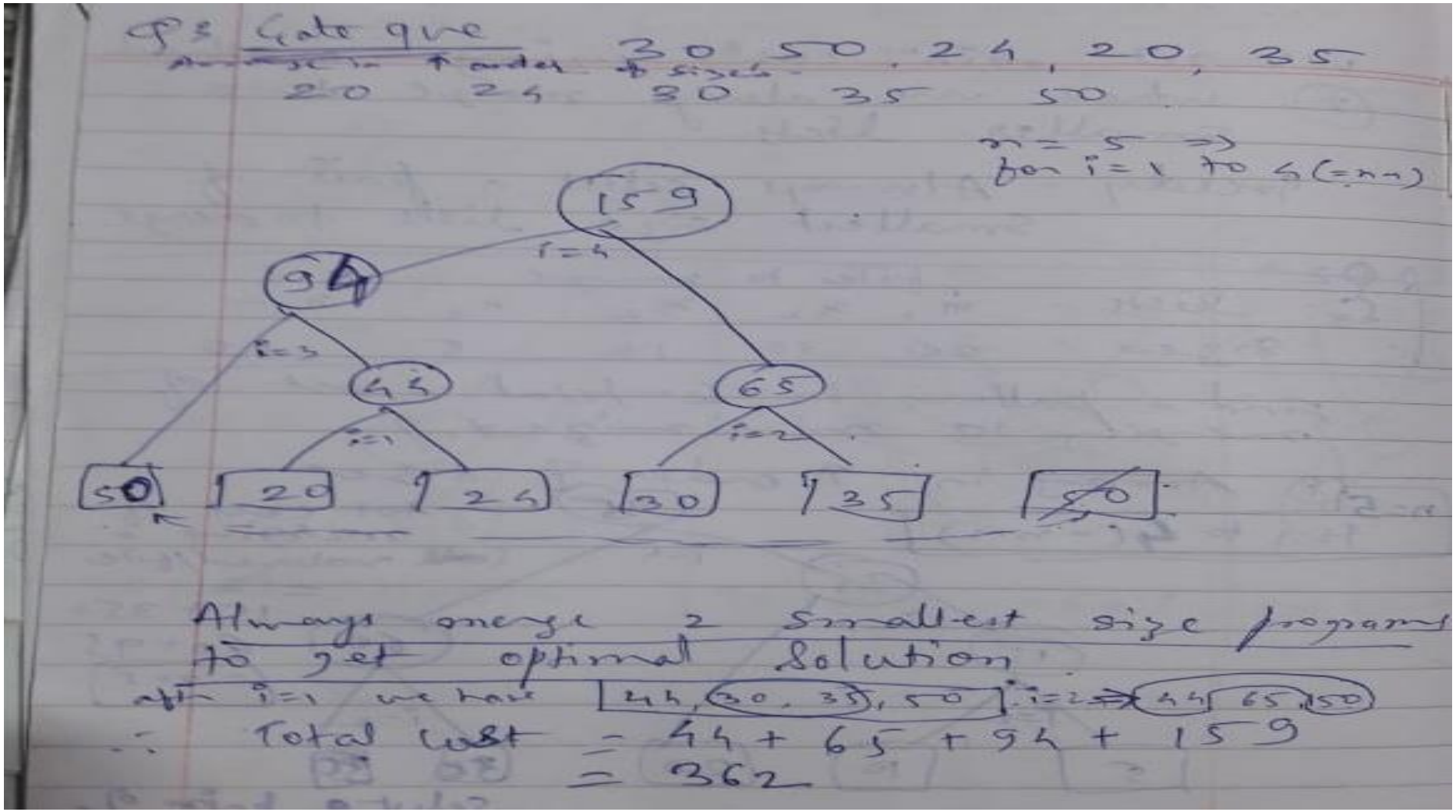
The solution takes :

8 + 15 + 21 + 29 + 32 + 50 + 82 =

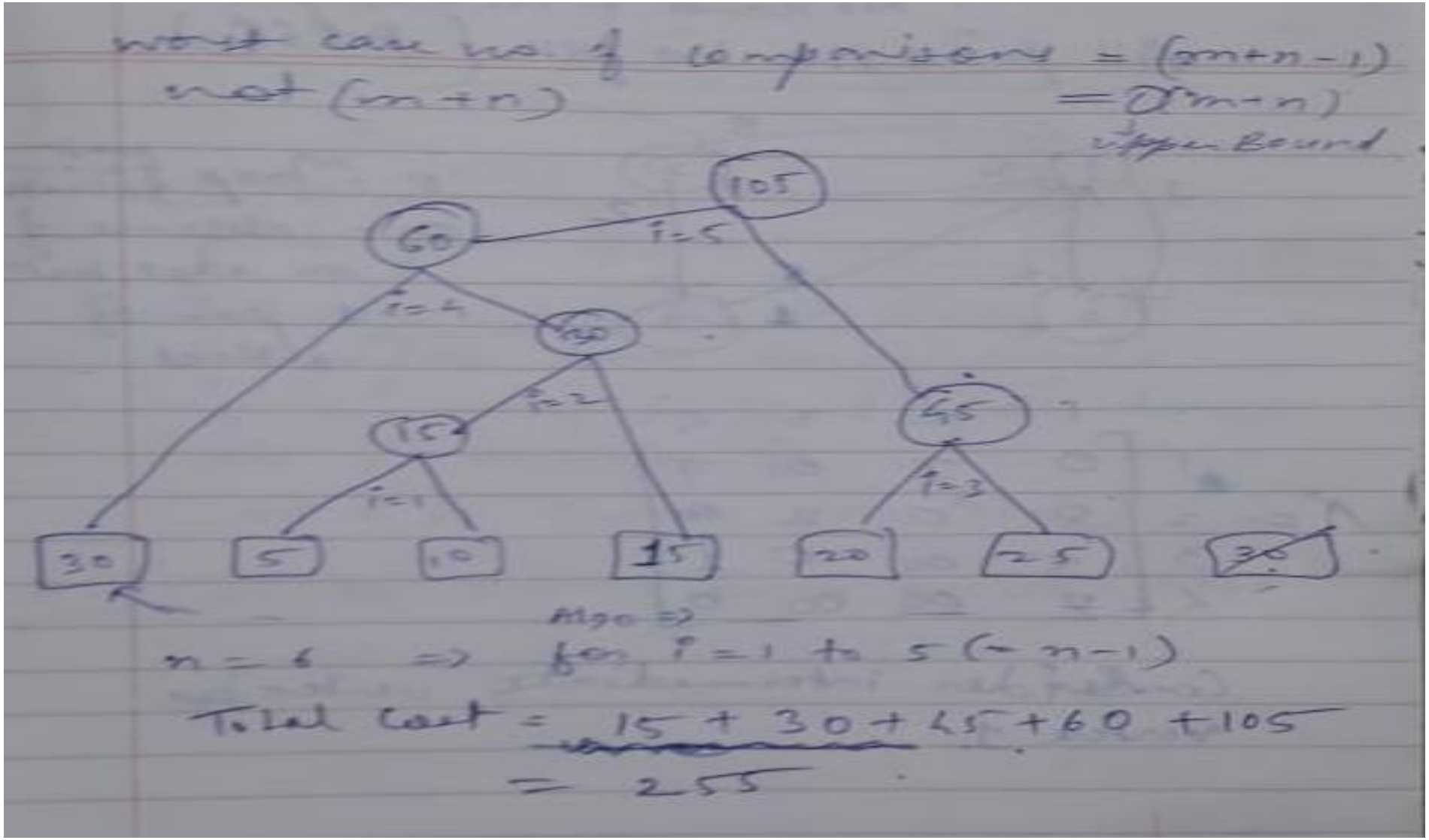
187 number of comparisons.

Ex2 : Merge the files with following lengths using Optimal Merge Pattern.
30, 5, 15 , 25, 20, 10.

Q1. **Find Optimal merge pattern for the given files, f_1, f_2, f_3, f_4 and f_5 with 30, 50, 24, 20 and 35 number of elements respectively.**



Q2. **Find Optimal merge pattern for the given files, f_1, f_2, f_3, f_4 and f_5, f_6, f7with 3, 5, 7, 9, 12, 14, 15 and 17 number of elements respectively.**



Contents of Module 3 : Greedy Algorithms :

1. Introduction
2. Knapsack problem – Algorithm & Time Complexity
3. Job sequencing with deadlines
5. Optimal storage on tapes
6. Optimal merge pattern
7. Analysis of All problems

Optimal Storage on Tapes :

Explain Optimal Storage on Tape with example (10M) (5 times)

Given n programs to be stored on a computer tape and

length of each program i is Li where 1<=i<=n,

Find the order in which the programs should be stored in the tape

for which the Mean Retrieval Time (MRT) given as below is minimized.

$$\frac{1}{n}\sum_{i=1}^{i=n}\sum_{j=1}^{j=i} L_j$$

A magnetic tape provides only sequential access of data.

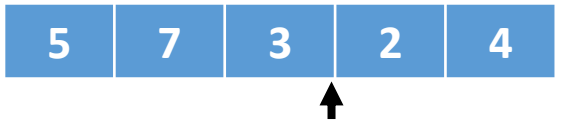
In an audio tape/cassette, unlike a CD, a fifth song from the tape can't be just directly played.

The length of the first four songs must be traversed to play the fifth song.

So in order to access certain data, head of the tape should be positioned accordingly.

Now suppose there are 5 songs in an audio tape with lengths 5, 7, 3, 2 and 4 min.s respectively.

In order to play the fourth song, we need to traverse an audio length of 5 + 7 + 3 = 15 mins and then position the tape head to the beginning of the fourth song.



Retrieval time of the data is the time taken to retrieve/access that data in its entirety.

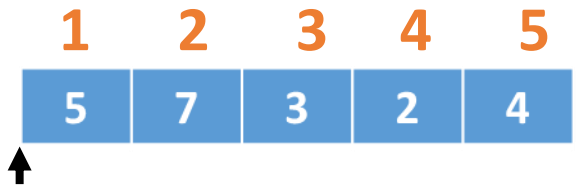
Hence retrieval time of the fourth song is 15 + 2 = 17 mins.

Optimal Storage on Tapes :

Now, considering that all programs in a magnetic tape are retrieved equally often and the tape head points to the front of the tape every time, a new term can be defined called the Mean Retrieval Time (MRT).

Let's suppose that the retrieval time of program i is Ti. Therefore, $T_i = \sum_{j=1}^{i} L_j$

MRT is the average of all such Ti.



Therefore $MRT = 1/n \sum_{i=1}^{n} T_i$ **, or**

$$MRT = 1/n \sum_{i=1}^{i=n} \sum_{j=1}^{j=i} L_j$$

Optimal Storage on Tapes :

The sequential access of data in a tape has some limitations.

Order must be defined in which the data/programs in a tape are stored so that least MRT can be obtained.

Hence the order of storing becomes very important to reduce the data retrieval/access time.

Thus, the task gets reduced – to define the correct order and hence minimize the MRT, i.e. to minimize the term

$$\sum_{i=1}^{n} \sum_{j=1}^{i} L_j$$

For e.g. Suppose there are 3 programs of lengths 2, 5 and 4 respectively.

2	5	4

So there are total 3! = 6 possible orders of storage on a single tape.

SR.NO.	ORDER	TOTAL RETRIEVAL TIME	MEAN RETRIEVAL TIME (MRT)
1	1 2 3	2 + (2 + 5) + (2 + 5 + 4) = 20	20/3
2	1 3 2	2 + (2 + 4) + (2 + 4 + 5) = 19	19/3
3	2 1 3	5 + (5 + 2) + (5 + 2 + 4) = 23	23/3
4	2 3 1	5 + (5 + 4) + (5 + 4 + 2) = 25	25/3
5	3 1 2	4 + (4 + 2) + (4 + 2 + 5) = 21	21/3
6	3 2 1	4 + (4 + 5) + (4 + 5 + 2) = 24	24/3

Observation : We get Min MRT when programs are sorted in the increasing order of their length.

Q. Store files of lengths (in MB) {12,34,56,73,24,11,34,56,78,91,34,91,45} on three tapes.

First sort the files in increasing order of their length (using either heap sort, merge sort or quick sort algo.)

1	2	3	4	5	6	7	8	9	10	11	12	13
11	12	24	34	34	34	45	56	56	73	78	91	91

Tape T0	11	34	45	73	91				
Tape T1	12	34	56	78					
Tape T2	24	34	56	91					

So this greedy method requires us to just sort the lengths of the programs i.e. arrange the programs in a non-decreasing order, so that they can be directly written into disk on their length basis.

MRT (T0) = [11 + (11+34) + (11+34+45) + (11+34+45+73) + (11+34+45+73+91)]/5 =
MRT (T1) = [12 + (12+34) + (12+34+56) + (12+34+56+78)]/4=
MRT (T2) = [24 + (24+34) + (24+34+56) + (24+34+56+91)]/4=

Greedy solution:

i. Make tape empty

ii. For i = 1 to n do;

iii. Select the next smallest program

iv. Put it on next tape.

The algorithm takes the best shortest term choice without checking whether it is the best long term decision.

Steps :

1. Sort the programs in increasing order of their lengths
2. Call Optimal Storage(n,m) algorithm

// n is the numbers of programs and
m is the numbers of tapes.

```
Algorithm Optimal_Storage (n, m)
{
K = 0; // Next tape to be stored.
For i = 1 to n do
    {
    Write (i, k); // "Assign program", j, "to tape", k;
    k = (k +1) mod m;
    }
}
```

k=0

For I = 1 to n do

	Tape(k)
Write (I, k)	0
k = (k+1) mod m	
1 mod 3 = 1	1
2 mod 3 = 2	2
---	---
3 mod 3 = 0	0
1 mod 3 = 1	1
2 mod 3 = 2	2
---	---
3 mod 3 = 0	0
1 mod 3 = 1	1
2 mod 3 = 2	2

Assign program I to tape k.

So, the complexity to sort the length of the programs = O(n logn).

Time required to write n programs on tape based on their length = O(n). (for i=1 to n)

So, Time complexity of optimal storage on tape algorithm = O(n log n) + O(n) = O(n logn).

Q. Find an optimal placement for 13 programs on 3 tapes T0, T1 & T2 where the program are of lengths 12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10 and 6.

1	2	3	4	5	6	7	8	9	10	11	12	13
3	4	5	5	6	7	8	10	11	12	18	26	32

Tape T0	3	5	8	12	32				
Tape T1	4	6	10	18					
Tape T2	5	7	11	26					

MRT (T0) = [11 + (11+34) + (11+34+45) + (11+34+45+73) + (11+34+45+73+91)]/5 =

MRT (T1) = [12 + (12+34) + (12+34+56) + (12+34+56+78)]/4=

MRT (T2) = [24 + (24+34) + (24+34+56) + (24+34+56+91))]/4=

End of Module 3
Thank You