Greedy Algorithms-II

Kumkum Saxena

Job Sequencing with Deadlines

- Problem Statement

- In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.(maximization problem).

- Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline.
- It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Points to remember

- In this problem we have n jobs j1, j2, … jn each has an associated deadline d1, d2, … dn and profit p1, p2, ... pn.
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled or processed at any given time.

- Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

- Output: Following is maximum
- profit sequence of jobs
- c, a

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
A	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum

profit sequence of jobs

c, a, e

- A Simple Solution is to generate all subsets of given set of jobs and check individual subset for feasibility of jobs in that subset.
- Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.

Job Sequencing with Deadlines

- Given n jobs. Associated with job I is an integer deadline $D_i \geqq 0$. For any job I the profit P_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time.
- A feasible solution is a subset J of jobs such that each job in the subset can be completed by its deadline. We want to maximize the

$$\sum_{i \in J} P_i$$

$n = 4, (p_1, p_2, p_3, p_4) = (100,10,15,27)$
$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

	Feasible solution	Processing sequence	value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3, 1	115
3	(1,4)	4, 1	127
4	(2,3)	2, 3	25
5	(3,4)	4,3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

This is a standard Greedy Algorithm problem. Following is algorithm.

- Sort all jobs in decreasing order of profit.
- Initialize the result sequence as first job in sorted jobs.
- Do following for remaining n-1 jobs
  - a) 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-Deadline (D, J, n, k)
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
        r := k
    while D(J(r)) > D(i) and D(J(r)) ≠ r do
        r := r – 1
    if D(J(r)) ≤ D(i) and D(i) > r then
        for l = k … r + 1 by -1 do
            J(l + 1) := J(l)
        J(r + 1) := i
        k := k + 1
```

Complexity Analysis

- In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is O(n^2),where n is the number of jobs.

- Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J1	J2	J3	J4	J5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J2	J1	J4	J3	J5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J2, as it can be completed within its deadline and contributes maximum profit.

Next, J1 is selected as it gives more profit compared to J4.

In the next clock, J4 cannot be selected as its deadline is over, hence J3 is selected as it executes within its deadline.

The job J5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J2, J1, J3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is 100 + 60 + 20 = 180.

Pseudocode

```
for i = 1 to n do
  Set k = min(dmax, DEADLINE(i))  //where DEADLINE(i) denotes deadline of ith job
  while k >= 1 do
        if timeslot[k] is EMPTY then
                    timeslot[k] = job(i)
              break
    endif
    Set k = k - 1
  endwhile
endfor
```

Optimal Storage on Tapes

- There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length L_i.
- Assume the tape is initially positioned at the front. If the programs are stored in the order
- $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j

$$t_j = \sum_{k=1}^{j} L_{i_k}$$

- Given n programs stored on a computer tape and length of each program i is L_i 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 $\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} L_j$
-) is minimized.

- Let us first break down the problem and understand what needs to be done.
- 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 4 songs in a tape of audio lengths 5, 7, 3 and 2 mins 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.
- 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.
- 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).

Optimal Storage on Tapes

If all programs are retrieved equally often, then the
mean retrieval time (MRT) = $\frac{1}{n}\sum_{j=1}^{n} t_j$

This problem fits the ordering paradigm.
Minimizing the MRT is equivalent to minimizing

$$d(I) = \sum_{j=1}^{n}\sum_{k=1}^{j} L_{i_k}$$

Example

Let n = 3, (L_1,L_2,L_3) = (5,10,3). 6 possible orderings. The optimal is 3,1,2

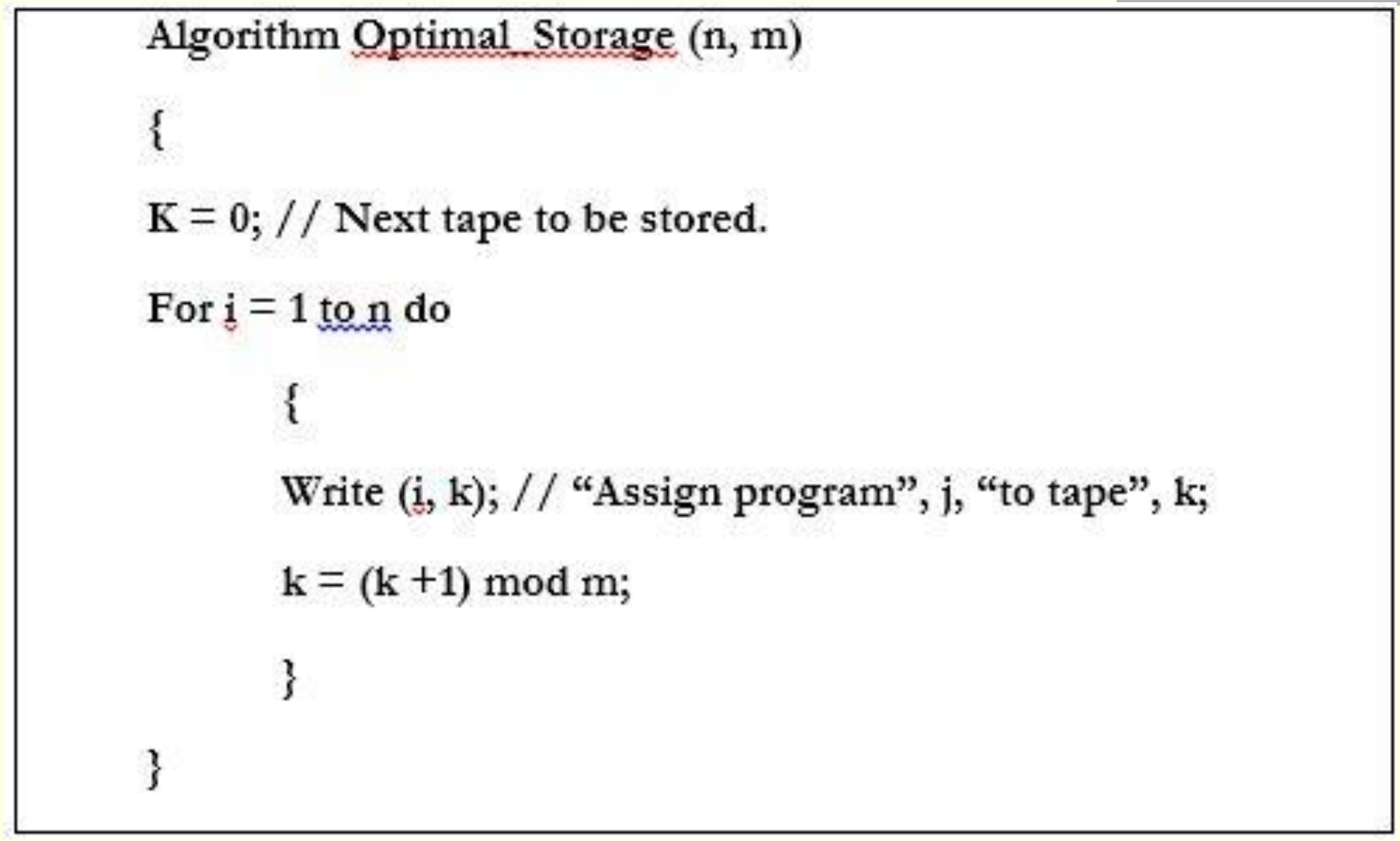
Ordering I	d(I)
1,2,3	5+5+10+5+10+3 = 38
1,3,2	5+5+3+5+3+10 = 31
2,1,3	10+10+5+10+5+3 = 43
2,3,1	10+10+3+10+3+5 = 41
3,1,2	3+3+5+3+5+10 = 29
3,2,1,	3+3+10+3+10+5 = 34

- For e.g. Suppose there are 3 programs of lengths 2, 5 and 4 respectively. So there are total 3! = 6 possible orders of storage.

Order		Total Retrieval Time	Mean Retrieval Time
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

- In above example, the first program’s length is added ‘n’ times, the second ‘n-1’ times…and so on till the last program is added only once.
- So, careful analysis suggests that in order to minimize the MRT, programs having greater lengths should be put towards the end so that the summation is reduced.
- Or, the lengths of the programs should be sorted in increasing order.
- That’s the Greedy Algorithm in use – at each step we make the immediate choice of putting the program having the least time first, in order to build up the ultimate optimized solution to the problem piece by piece.

Algorithm



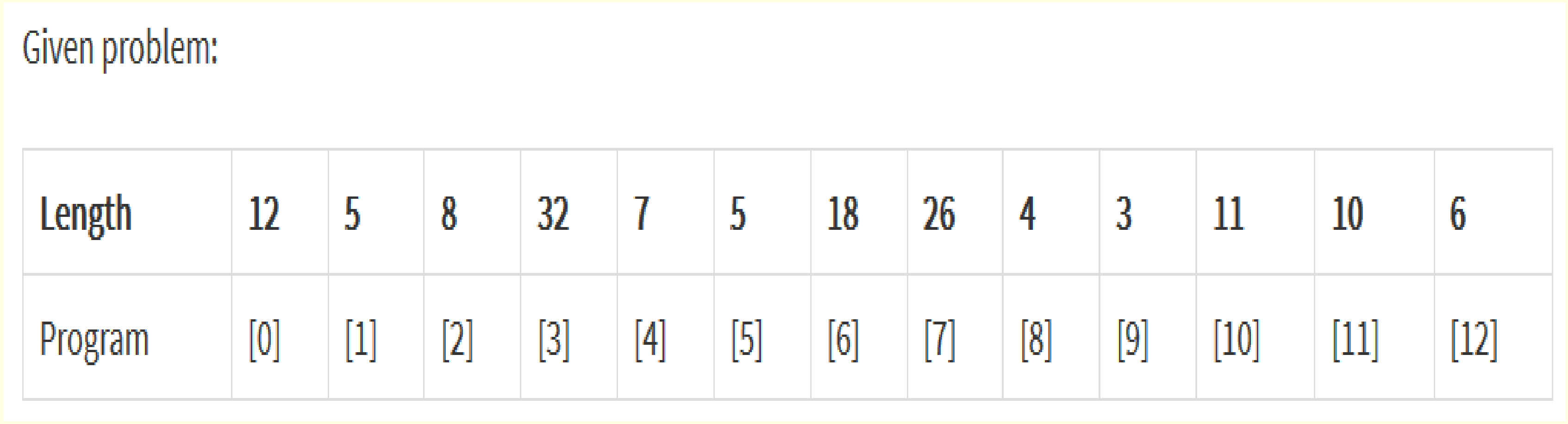
```
Algorithm Optimal_Storage (n, m)
{
K = 0; // Next tape to be stored.
For i = 1 to n do
    {
    Write (i, k); // "Assign program", j, "to tape", k;
    k = (k +1) mod m;
    }
}
```

Time Complexity

- Time complexity of the above program is the time complexity for sorting,
- that is O(n log n) (Since std::sort() operates in O(n log n))
- If you use bubble sort instead of std::sort(), it will take O(n^2)

Examples

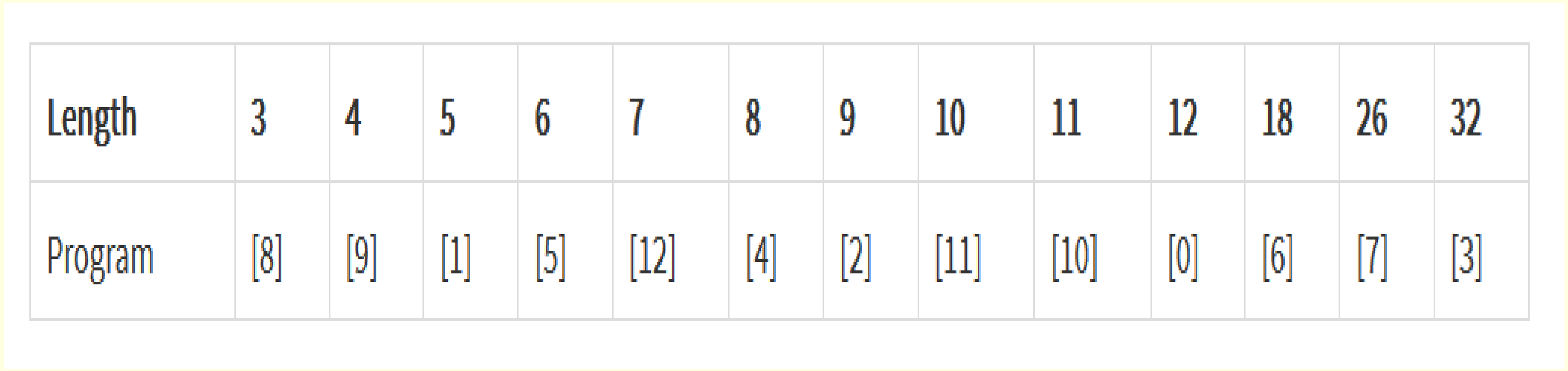
Find an optimal placement for 13 programs on 3 tapes T0, T1 & T2 where the program are of lengths 12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10 and 6.



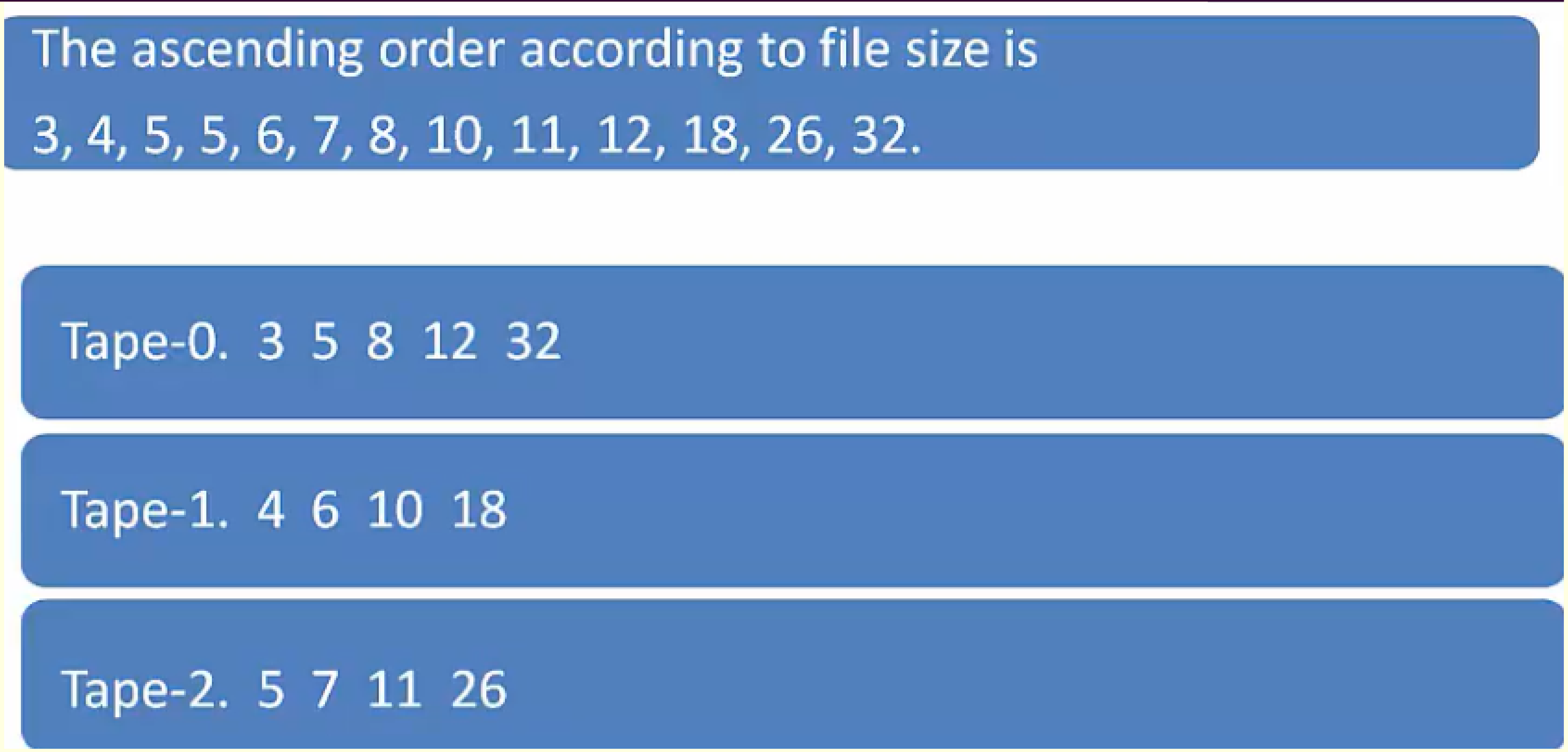
Given problem:

Length	12	5	8	32	7	5	18	26	4	3	11	10	6
Program	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

We organize the program as:



Length	3	4	5	6	7	8	9	10	11	12	18	26	32
Program	[8]	[9]	[1]	[5]	[12]	[4]	[2]	[11]	[10]	[0]	[6]	[7]	[3]



The ascending order according to file size is

3, 4, 5, 5, 6, 7, 8, 10, 11, 12, 18, 26, 32.

Tape-0. 3 5 8 12 32

Tape-1. 4 6 10 18

Tape-2. 5 7 11 26

```
void findOrderMRT(int L[], int n)
{
    // Here length of i'th program is L[i]
    sort(L, L + n);
    // Lengths of programs sorted according to increasing
   // lengths. This is the order in which the programs
    // have to be stored on tape for minimum MRT
    cout << "Optimal order in which programs are to be stored is: ";
    for (int i = 0; i < n; i++)
        cout << L[i] << " ";
    cout << endl;
    // MRT - Minimum Retrieval Time
    double MRT = 0;
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = 0; j <= i; j++)
            sum += L[j];
        MRT += sum;
    }
    MRT /= n;
    cout << "Minimum Retrieval Time of this order is " << MRT;
}
```

Optimal Merge Pattern

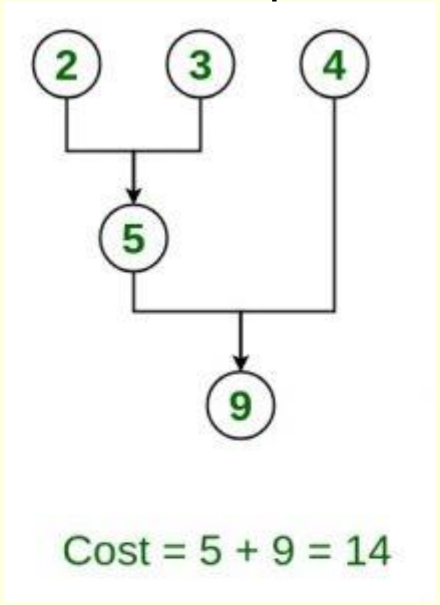
Optimal merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.
If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.
There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time. The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

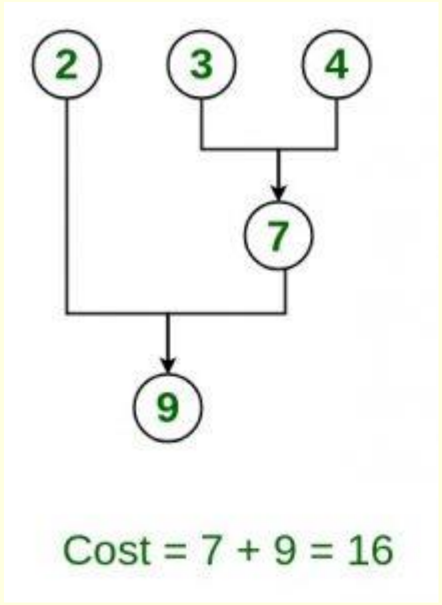
Where, f (i) represents the number of records in each file and d (i) represents the depth.

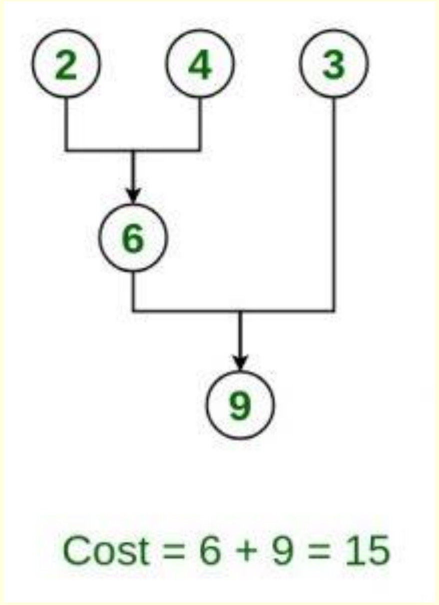
$$\sum_{i=1}^{n} f(i)d(i)$$

Given 3 files with size 2, 3, 4 units.Find optimal way to combine these files.
Input: n = 3, size = {2, 3, 4}
Explanation: There are different ways to combine these files:

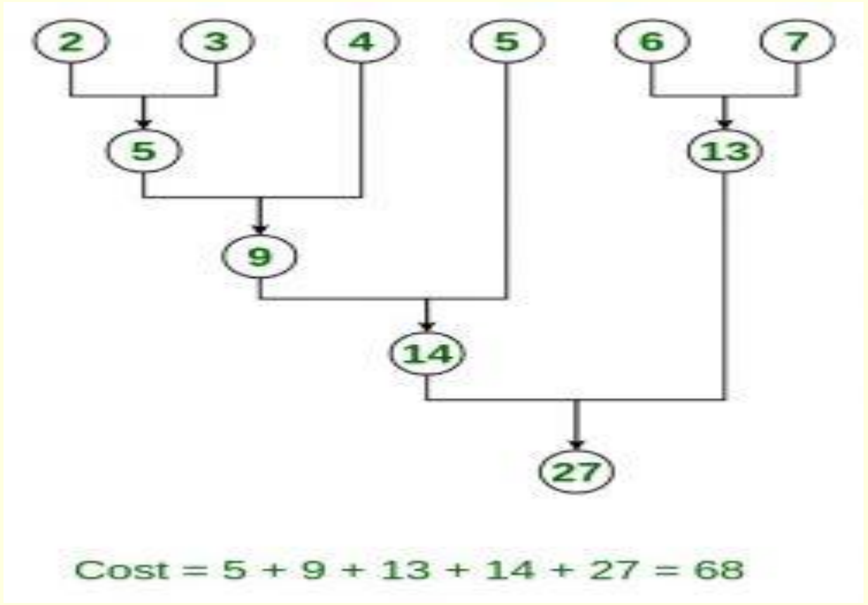
Method 1: Optimal method **Method 2:** **Method 3:**







Input: n = 6, size = {2, 3, 4, 5, 6, 7}
Output: 68
Explanation: Optimal way to combine these files



Optimal merge Pattern

Given n number of sorted files, the task is to find the minimum computations done to reach Optimal Merge Pattern.
When two or more sorted files are to be merged all together to form a single file, the minimum computations done to reach this file are known as **Optimal Merge Pattern**.
If more than 2 files need to be merged then it can be done in pairs. For example, if need to merge 4 files A, B, C, D. First Merge A with B to get X1, merge X1 with C to get X2, merge X2 with D to get X3 as the output file.
If we have two files of sizes m and n, the total computation time will be m+n. Here, we use greedy strategy by merging two smallest size files among all the files present.

Approach

Node represents a file with a given size also given nodes are greater than 2

```
1.Add all the nodes in a priority queue (Min Heap).{node.weight = file size}
2.Initialize count = 0 // variable to store file computations.
3.Repeat while (size of priority Queue is greater than 1)
        1.create a new node
        2.new node = pq.poll().weight+pq.poll().weight;
        //pq denotes priority queue, remove 1st smallest and 2nd smallest element
         and add their weights to get a new node
        3.count += node.weight
        4.add this new node to priority queue;
4.count is the final answer
```

Algorithm for optimal merge pattern

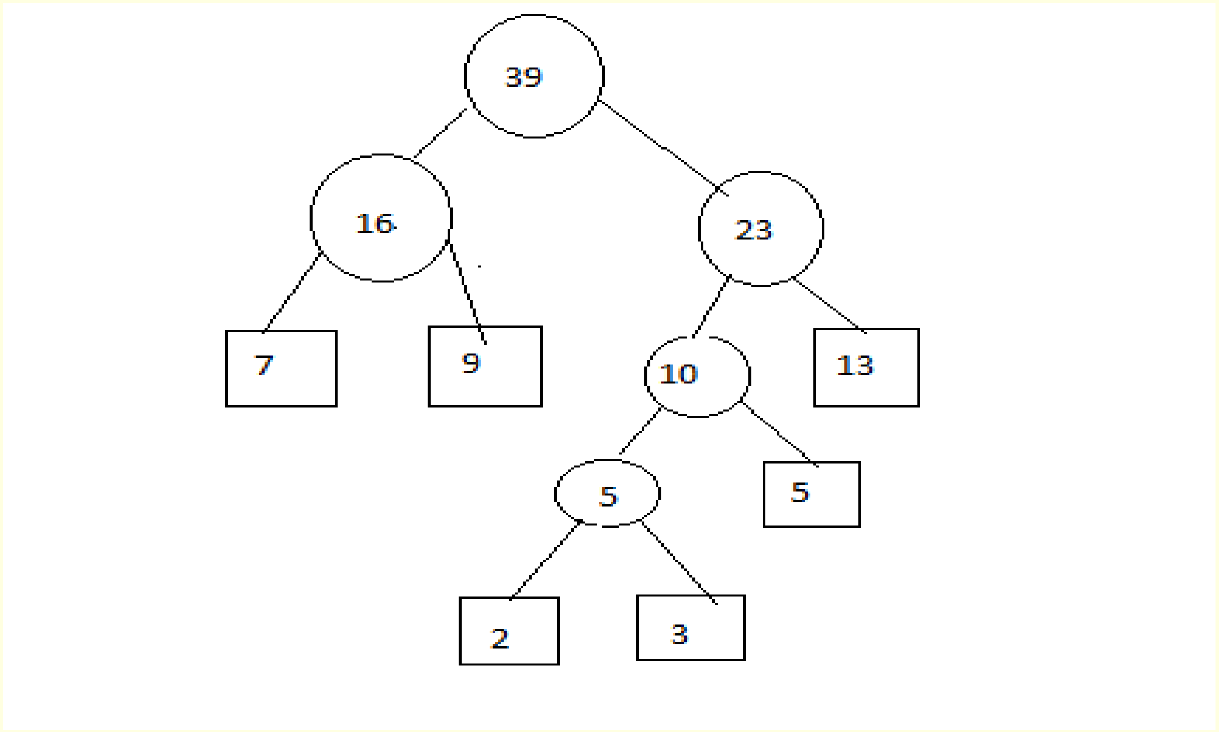
```
Algorithm Tree(n)
 //list is a global list of n single node
 {
 for i=1 to i= n-1 do
 {
            // get a new tree node
             Pt=new treenode;
            // merge two trees with smallest length
            (Pt = lchild) = least(list);
            (Pt = rchild) = least(list);
            (Pt =weight) = ((Pt = lchild) = weight) = ((Pt = rchild) = weight);
             Insert (list , Pt);
 }
// tree left in list
Return least(list);
}
```

- An optimal merge pattern corresponds to a binary merge tree with minimum weighted external path length.
- The function tree algorithm uses the greedy rule to get a two- way merge tree for n files. The algorithm contains an input list of n trees.
- There are three field child, rchild, and weight in each node of the tree.
- Initially, each tree in a list contains just one node.
- This external node has lchild and rchild field zero whereas weight is the length of one of the n files to be merged.
- For any tree in the list with root node t, t = it represents the weight that gives the length of the merged file.
- There are two functions least (list) and insert (list, t) in a function tree.
- Least (list) obtains a tree in lists whose root has the least weight and return a pointer to this tree.
- This tree is deleted from the list. Function insert (list, t) inserts the tree with root t into the list.

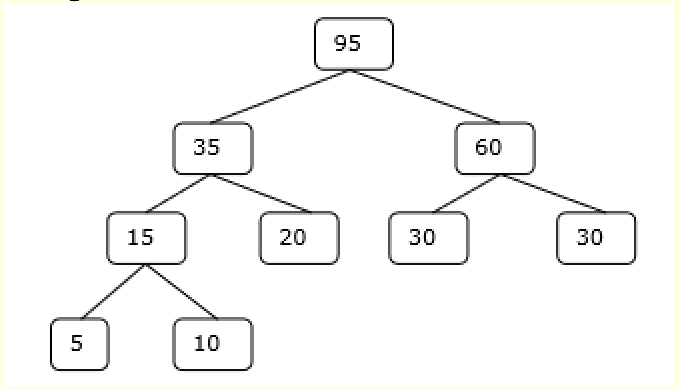
- The main for loop in this algorithm is executed in n-1 times.
- If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only O(1) time and insert (list, t) can be performed in O(n) time.
- Hence, the total time taken is O (n2).
- If the list is represented as a minheap in which the root value is less than or equal to the values of its children, then least (list) and insert (list, t) can be done in O (log n) time.
- In this condition, the computing time for the tree is O (n log n).

Example:
Given a set of unsorted files: 5, 3, 2, 7, 9, 13
Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13
After this, pick two smallest numbers and repeat this until we left with only one number.

So, The merging cost = 5 + 10 + 16 + 23 + 39 = 93



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.



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

Analysis

- A greedy algorithm typically makes (approximately) n choices for a problem of size n
    - (The first or last choice may be forced)
- Hence the expected running time is:
- O(n * O(choice(n))), where choice(n) is making a choice among n objects
    - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an O(k)=O(1) operation;
        - Therefore, coin counting is (n)
    - Huffman: Must sort n values before making n choices
        - Therefore, Huffman is O(n log n) + O(n) = O(n log n)
    - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is O(n log n) overall
        - Therefore, MST is O(n log n) + O(n) = O(n log n)