U N I T - 4

Greedy Method : The General Method – Optimal Storage on Tapes – Knapsack Problem – Job Sequencing with Deadlines – Optimal Merge Patterns.

The Greedy Method - The General Method

The greedy method is the most straightforward design technique. It can be applied to a wide variety of problems.

Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints.

Any subset that satisfies these constraints is called a feasible solution.

We need to find a feasible solution that either maximizes or minimizes a given objective function.

A feasible solution that does this is called an optimal solution.

The greedy method suggests that one can devise an algorithm that worksin stages, considering one input at a time.

At each stage, a decision is made regarding whether a particular input is in an optimal solution.

This is done by considering the inputs in an order determined by some selection procedure.

If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added.

The selection procedure itself is based on some optimization measure. This measure may be the objective function. In fact, several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the subset paradigm.

The function Select selects an input from a[] and removes it. The selected input's value is assigned to x.

Feasible is a Boolean-valued function that determines whether x can be included into the solution vector.

The function Union combines x with the solution and updates the objective function.

The function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

```
Algorithm Greedy(a, n)
// a[l : n] contains the n inputs.
{
    solution := 0; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution:= Union(solution, x);
    }
    return solution;
}
```

<u>KNAPSACK PROBLEM</u>

We are given n objects and a knapsack or bag.

Object i has a weight Wi and the knapsack has a capacity m.

If a fraction x_i $0\leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of P_iX_i is earned.

The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.

Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

The profits and weights are positive numbers. A feasible solution is any set (x_1, ... , Xn) satisfying the given constraints. An optimal solution is a feasible solution which is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, **solution 4** yields the maximum profit. This solution is optimal for the given problem instance.

```
Algorithm GreedyKnapsack (m, n)
{
for i→1 to n do  x[i] = 0.0;
U = m;
for i→1 to n do
 {
        if(w[i] > U)  then break;
        x[i] = 1.0;
        U = U – w[i];
        }
        if (i<=m) x[i] = U/w[i];
}
```

JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $P_i \geq \mathbf{0}$.

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. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.

The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \varepsilon J} p_i$. An optimal solution is a feasible solution with maximum value.

Example: Let n = 4, (p1,P2,p3,p4) = (100, 10, 15, 27) and (d1, d2, d3, d4) = (2, 1, 2, 1). The feasible solutions and their values are:

	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

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1is completed at time 2.

```
Algorithm GreedyJob($d, J, n$)
// $J$ is a set of jobs that can be completed by their deadlines.
{
    $J := \{1\}$;
    for $i := 2$ to $n$ do
    {
        if (all jobs in $J \cup \{i\}$ can be completed
            by their deadlines) then $J := J \cup \{i\}$;
    }
}
```

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, $1 \le i \le n$. All programs can be stored on the tape if and only if the sum of the lengths of the programs is at most l.

We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front.

Hence, 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 is proportional to $\sum_{1 \le k \le j} l_{ik}$.

If all programs are retrieved equally often, then the expected or mean retrieval time (MRT) is $(1/n) \sum_{1 \le j \le n} t_j$.

In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized.

This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{ik}$.

Example: Let n = 3 and $(l_i, l_2, l_3) = (5, 10, 3)$. There are n! = 6 possible orderings. These orderings and their respective d values are:

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

The optimal ordering is 3, 1**,** 2. □

A greedy approach to building the required permutation would choose the next program on the basis of some optimization measure. The next program to be stored on the tape would be one that minimizes the increase in d. We observe that the increase in d is minimized if the next program chosen is the one with the ***least length*** from among the remaining programs.

```
Algorithm Store(n, m)
// n is the number of programs and m the number
of tapes.
{
    j := O; // Next tape to store on
    for i := 1 to n do
    {
        write ("append program", i,
            "to permutation for tape", j);
        j := (j +1) mod m;
    }
}
```

OPTIMAL MERGE PATTERNS

The optimal merge pattern problem describes that there two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time O(n +m).

When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs.

Thus, if files x_1 , x_2, x_3 and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then we could merge y_1 and x_3 to get y_2. Finally, wecould merge y_2 and x_4 to get the desired sorted file.

Alternatively, we could first merge x_1 and x_2 getting y_l, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 and get the desired sorted file.

Given n sorted files, there are many ways in which to pairwise merge them into a single sortedfile.

Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal wayto pairwise merge n sorted files.

Example: Method 1: The files x_1, x_2 and x_3 are three sorted files of length 30, 20, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110.

Method 2: Merge x_2 and X_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90.

Hence, the second merge pattern is faster than the first.

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n-record file and an m-record file requires possibly n +m record moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together.

Thus, if we have five files (x_1,... , x_5) with sizes (20, 30, 10, 5, 30), our greedy rule would generate the following merge pattern:

1. Merge x_4 and x_3 to get z1 (15 moves)
2. Merge z_1 and x_1 to get z_2 (35 moves)
3. Merge x_2 and x_5 to get z_3 (60 moves)
4. Merge z_2 and z_3 to get the answer z_4 (95 moves)

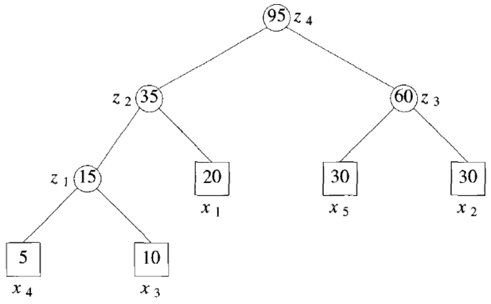
The total number of record moves is 205.

The merge pattern such as the one just described will be referred to as a two-way merge pattern (each merge step involves the merging of two files). The two-way merge patterns can be represented by binary merge trees.

Figure shown below a binary merge tree representing the optimal merge pattern obtained for the above five files.

The leaf nodes are drawn as squares and represent the given five files. These nodes are called external nodes.

The remaining nodes are drawn as circles and are called internal nodes. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number of records) of the file represented by that node.



The external node x_4 is at a distance of 3 from the root node z_4 (a node at level i is at a distance of i - 1 from the root). Hence, the records of file $x4$ are moved three times, once to get z_1, once again to get z_2, and finally one more time to get z_4. If d_i is the distance from the root to the external node for file x_i and q_i, the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i$$

This sum is called the *weighted external path length* of the tree.

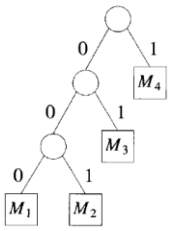
```
treenode = record {
      treenode* !child; treenode* rchild;
      integer weight; };
Algorithm Tree(n)      {
      for i := 1 to n - 1 do         {
            pt:= new treenode; // Get a new tree node.
            (pt→[child) := Least(list); // Merge two trees with
            (pt→rchild) := Least(list); // smallest lengths.
            (pt→weight) := ((pt→lchild) →weight)
                        + ((pt→rchild) →weight);
            Insert(list, pt);
      }
      return Least(list); // Tree left in list is the merge tree.
    }
```

Huffman Codes

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages M1, ... , Mn+l·

Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree.

A decode tree is a binary tree in which external nodes represent messages.



The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node.

For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of the above Figure corresponds to codes 000, 001, 01, and 1 for messages M1, M2, M3, and M4 respectively. These codes are called Huffman codes.