**UNIT 2**

**Greedy Method**

Greedy Method is one of the methods used to develop algorithms or solve problems. Any problem that can be solved using Greedy method has number of inputs. A solution of the problem is generated by selecting some inputs from all inputs of the problem. There exist number of solutions for the problems that can be solved using greedy method. These solutions are called feasible solutions. Out of these feasible solutions, we need to select the best solution based on an objective function. The best solution is called an optimal solution.

Ex: Customer went to a shop and purchased goods for 48/- and gave 100/- to the shop keeper and the shop keeper has to return 52/-

inputs: 2000, 500, 200, 100, 50, 20, 10, 5, 2, 1(currency notes)

feasible solutions:

<50, 2>

<50, 1, 1>

<20, 20, 10, 2>

.

.

.

Optimal solution: <50, 2>

Objective function: returning change by using the minimum number of currency notes

**Control Abstraction of Greedy Method**

The following algorithm indicates the procedure to generate a feasible solution of the problem using greedy method.

Algorithm Greedy(a, n)

// a[1:n] is an array containing inputs of the problem

{

solution := {};

for i := 1 to n do

{

x := Select(a);

if Feasible(x) then

{

solution := union(solution, x);

}

}

return solution;

}

**Select()** is a user defined function which selects an input from array ‘a’

**Feasible()** is a user defined function which decides whether input x can be included in the solution or not

**Applications of Greedy method**

1. Knapsack problem
2. Single Source Shortest Path problem
3. Optimal Merge Patterns problem
4. Optimal Storage on Tapes problem

**Knapsack problem**

There is a bag with capacity m. There are n number of items. These items are placed one by one into the bag. Items can be placed into the bag in any order. Each item has a weight wi and a profit pi. When an item is placed into the bag then the capacity of bag is decremented by the weight of item and the corresponding profit is earned. An item can be placed into the bag either fully or partially. Part of item placed into the bag is indicated by xi. We need to place items into the bag such that we get maximum profit and the total weight of items should not exceed the capacity of bag.

Ex: Find optimal solution of the following knapsack problem

Capacity of bag m=20

Number of items n=3

weights of items (w1, w2, w3) = (18, 15, 10)

profits of items (p1,p2, p3) = (25, 20, 15)

The procedure for finding optimal solution of the knapsack problem is as follows:

1. Calculate per unit profit of items pi/wi:

25/18, 20/15, 15/10 = 1.38, 1.33, 1.5

2. Place items into bag in descending order of per unit profit of items

3, 1, 2

3. m=20 x3=1 w3\*x3=10\*1=10 p3\*x3=15\*1=15

m=20-10=10 x1=10/18 w1\*x1=18\*10/18=10 p1\*x1=25\*10/18=13.8

m=10-10=0 x2=0

Optimal solution is: <x1, x2, x3>=<10/18, 0, 1> Profit=28.8

Ex: Find optimal solution of the following knapsack problem

m=10 n=7 (w1, w2, w3, w4, w5, w6, w7) = (1, 4, 3, 2, 3, 6, 7)

(p1,p2, p3, p4,p5, p6, p7) = (7, 18, 6, 7, 5, 3, 4)

1. Calculate per unit profit of items pi/wi:

7/1, 18/4, 6/3, 7/2, 5/3, 3/6, 4/7 = 7, 4.5, 2, 3.5, 1.67, 0.5, 0.57

2. Place items into bag in descending order of per unit profit of items

1, 2, 4, 3, 5, 7, 6

3. m=10 x1=1 w1\*x1=1\*1=1 p1\*x1=7\*1=7

m=10-1=9 x2=1 w2\*x2=4\*1=4 p2\*x2=18\*1=18

m=9-4=5 x4=1 w4\*x4=2\*1=2 p4\*x4=7\*1=7

m=5-2=3 x3=1 w3\*x3=3\*1=3 p3\*x3=6\*1=6

m=3-3=0 x5=0

x7=0

x6=0

Optimal solution < x1,x2, x3, x4,x5, x6, x7>=<1,1,1,1,0,0,0> profit=38

**Algorithm**

Algorithm Knapsack(n, m, p, w, x, profit)

//n is number of items

//m is capacity of bag

//p[1:n] is an array containing profits of items

//w[1:n] is an array containing weights of items

//x[1:n] is an array used to store optimal solution of the problem

//profit is used to store profit of the optimal solution

{

profit := 0;

for i := 1 to n do

{

x[i] := 0;

pp[i] := p[i]/w[i];

}

for i := 1 to n-1 do

{

for j := i+1 to n do

{

if pp[i] < pp[j] then

{

swap(pp[i], pp[j]);

swap(p[i], p[j]);

swap(w[i], w[j]);

}

}

}

for i := 1 to n do

{

if w[i] > m then

break;

else

{

x[i] := 1;

m := m-w[i]\*x[i];

profit := profit+p[i]\*x[i];

}

}

x[i] := m/w[i];

profit := profit+p[i]\*x[i];

for i := 1 to n do

write x[i];

write profit;

}

Ex: Find optimal solution of the following knapsack problem

m=15 n=7 (w1, w2, w3, w4, w5, w6, w7) = (2, 3, 5, 7, 1, 4, 1)

(p1,p2, p3, p4,p5, p6, p7) = (10, 5, 15, 7, 6, 18, 3)

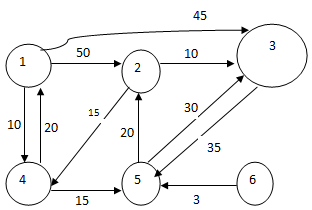
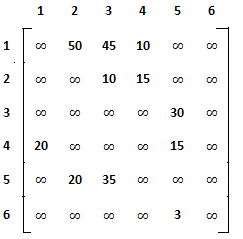
**Single Source Shortest Path Problem**

**Input**: Weighted directed graph

We have to select one node in the given graph as starting point or source node. Generally, the node with no incoming edges is selected as starting point. If there is no such node then any is selected as starting point. Then, we need to find shortest paths from starting point to all remaining nodes in the graph.

Ex: Find shortest paths in the following graph

Cost matrix/Adjacency matrix

Node 6 is selected as starting point as node 6 is not having incoming edges.

Shortest paths are identified from node 6 to all remaining nodes.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Set- s** | **Next node selected** | **dist[i]** | | | | | | **pr[i]** | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 |
| **initial** | {} | 6 |  |  |  |  | 3 |  | 6 | 6 | 6 | 6 | 6 | 0 |
| **1** | {6} | 5 |  | 23 | 38 |  | 3 |  | 6 | 5 | 5 | 6 | 6 | 0 |
| **2** | {6,5} | 2 |  | 23 | 33 | 38 | 3 |  | 6 | 5 | 2 | 2 | 6 | 0 |
| **3** | {6,5,2} | 3 |  | 23 | 33 | 38 | 3 |  | 6 | 5 | 2 | 2 | 6 | 0 |
| **4** | {6,5,2,3} | 4 | 58 | 23 | 33 | 38 | 3 |  | 4 | 5 | 2 | 2 | 6 | 0 |
| **5** | {6,5,2,3,4} | 1 | 58 | 23 | 33 | 38 | 3 |  | 4 | 5 | 2 | 2 | 6 | 0 |
| **6** | {6,5,2,3,4,1} | - | 58 | 23 | 33 | 38 | 3 |  | 4 | 5 | 2 | 2 | 6 | 0 |

**Shortest path from node 6**

**To node 1 To node 3 To node 5**

pr[1]=4 pr[3]=2 pr[5]=6

pr[4]=2 pr[2]=5

pr[2]=5 pr[5]=6 5🡨6

pr[5]=6

1🡨4🡨2🡨5🡨6 3🡨2🡨5🡨6

**To node 2 To node 4**

pr[2]=5 pr[4]=2

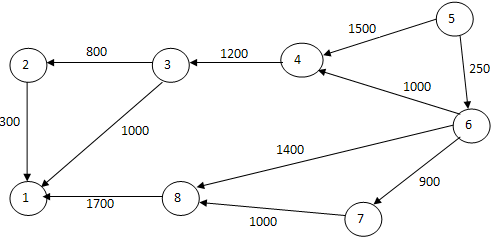
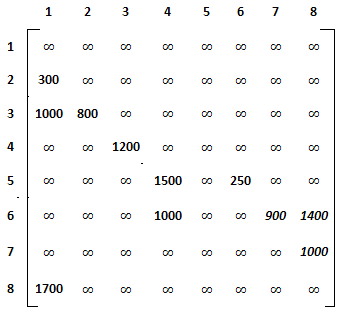
pr[5]=6 pr[2]=5

pr[5]=6

2🡨5🡨6

4🡨2🡨5🡨6

Ex: Find shortest paths in the following graph

Node 5 is selected as starting point as node 5 is not having incoming edges.

Shortest paths are identified from node 5 to all remaining nodes.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Set- s** | **Next node selected** | **dist[i]** | | | | | | | | **pr[i]** | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **initial** | {} | 5 |  |  |  | 1500 |  |  |  |  | 5 | 5 | 5 | 5 | 0 | 5 | 5 | 5 |
| **1** | {5} | 6 |  |  |  |  |  | 250 | 1150 | 1650 | 5 | 5 | 5 | 6 | 0 | 5 | 6 | 6 |
| **2** | {5,6} | 7 |  |  |  |  |  | 250 | 1150 | 1650 | 5 | 5 | 5 | 6 | 0 | 5 | 6 | 6 |
| **3** | {5,6,7} | 4 |  |  |  |  |  | 250 | 1150 | 1650 | 5 | 5 | 4 | 6 | 0 | 5 | 6 | 6 |
| **4** | {5,6,7,4} | 8 |  |  |  |  |  | 250 | 1150 | 1650 | 8 | 5 | 4 | 6 | 0 | 5 | 6 | 6 |
| **5** | {5,6,7,4,8} | 3 |  | 3250 |  |  |  | 250 | 1150 | 1650 | 8 | 3 | 4 | 6 | 0 | 5 | 6 | 6 |
| **6** | {5,6,7,4,8,3} | 2 |  | 3250 |  |  |  | 250 | 1150 | 1650 | 8 | 3 | 4 | 6 | 0 | 5 | 6 | 6 |
| **7** | {5,6,7,4,8,3,2} | 1 |  | 3250 |  |  |  | 250 | 1150 | 1650 | 8 | 3 | 4 | 6 | 0 | 5 | 6 | 6 |
| **8** | {5,6,7,4,8,3,2,1} | - |  | 3250 |  |  |  | 250 | 1150 | 1650 | 8 | 3 | 4 | 6 | 0 | 5 | 6 | 6 |

**Shortest path from node 5**

**To node 1 To node 3 To node 6**

pr[1]=8 pr[3]=4 pr[6]=5

pr[8]=6 pr[4]=6

pr[6]=5 pr[6]=5 6🡨5

1🡨8🡨6🡨5 3🡨4🡨6🡨5

**To node 2 To node 4 To node 7**

pr[2]=3 pr[4]=6 pr[7]=6

pr[3]=4 pr[6]=5 pr[6]=5

pr[4]=6

pr[6]=5 4🡨6🡨5 7🡨6🡨5

2🡨3🡨4🡨6🡨5

**To node 8**

pr[8]=6

pr[6]=5

8🡨6🡨5

Algorithm SSSP(n, cost, s)

//n is number of nodes in the given graph

//cost[1:n,1:n] is adjacency matrix of the given graph

//s is starting point or source node

{🡨

for i := 1 to n do

{

set[i] := false;

dist[i] := cost[s, i];

pr[i] := s;

}

u := s;

pr[s] := 0;

for j := 1 to n do

{\*

set[u] := true;

min := ∞;

for i := 1 to n do

{

if set[i] := false then

{

if dist[i]<min then

{

min := dist[i];

u := i;

}

}

}

for i := 1 to n do

{

if set[i] := false then

{

if dist[i]>dist[u]+cost[u,i] then

{

dist[i] := dist[u]+cost[u,i];

pr[i] := u;

}

}

}

\*}

for i := 1 to n do

{

if i ≠ s then

{

k := i;

write k;

while pr[k] ≠ s do

{

k := pr[k];

write (“🡨”, k);

}

write (“🡨”, s);

}

}

🡪}

**Optimal Merge Pattern Problem**

Input to Optimal Merge Pattern problem is n number of files. Each file contains number of values or records. We need to find the optimal merge pattern for merging the n number of files.

At a time two files are merged. A merge pattern indicates an order in which the files are merged. There exist n(n-1)/2 ways (merge patterns) for merging the n number of files. Two files (f1, f2) can be merged in one way

merge pattern: f1f2

Three files (f1, f2, f3) can be merged in three ways

merge pattern1: ((f1f2)f3)

merge pattern2: ((f1f3)f2)

merge pattern3: ((f2f3)f1)

Four files (f1, f2, f3, f4) can be merged in six ways.

Binary Merge Tree

A merge pattern is represented graphically through binary merge tree. In the binary merge tree, leaf nodes are called external nodes. External nodes represent the files. External nodes are indicated by boxes. Non leaf nodes are called internal nodes. Each internal node has exactly two child nodes. Each internal node represents a new file which is formed by merging two files. Internal nodes are indicated by circles.

One parameter called ‘Total movement of values’ is calculated for a merge pattern using the following formula:

Total movement of values =

Where n is number of files, di is distance of file i from root node, si is size of file i.

Optimal merge pattern for merging a list of files is identified based on the total movement of values. The merge pattern with least movement of values is selected as the optimal merge pattern.

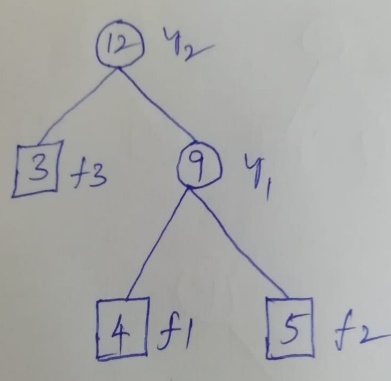
Ex: Consider the following three files

f1 with 4 values

f2 with 5 values

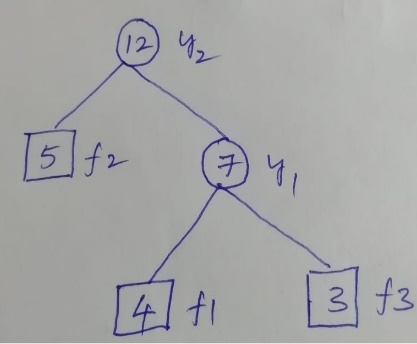
f3 with 3 values

Binary merge tree for merge pattern ((f1f2)f3) is



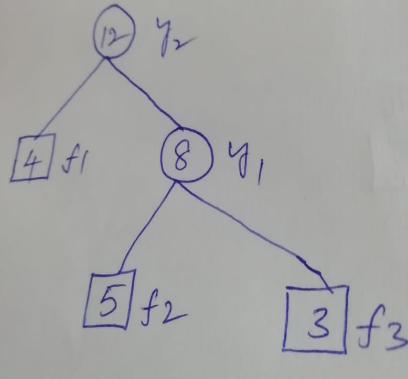
Total movement of values = 2\*4+2\*5+1\*3=8+10+3=21

Binary merge tree for merge pattern ((f1f3)f2) is



Total movement of values = 2\*4+1\*5+2\*3=8+5+6=19

Binary merge tree for merge pattern ((f2f3)f1) is



Total movement of values = 1\*4+2\*5+2\*3=4+10+6=20

In above example, ((f1f3)f2) is the optimal merge pattern.

**Procedure for finding the optimal merge pattern**

1. Create a tree for each file and include all created trees into a list ‘L’.
2. Repeat for n-1 times:

a) Select two trees (files) with least values at root (sizes) from the list L and merge them to get another tree (file).

b) Remove the selected two trees (files) from the list L.

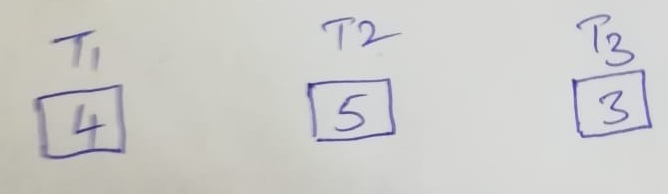
c) Include the new tree (file) into the list L.

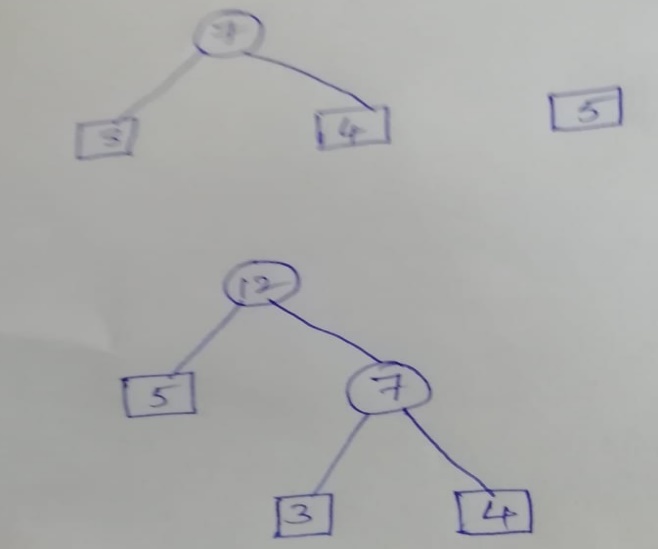
Ex: find optimal merge pattern for merging the following files

f1 with 4 values

f2 with 5 values

f3 with 3 values





The optimal merge pattern for merging the above three files is ((f1f3)f2).

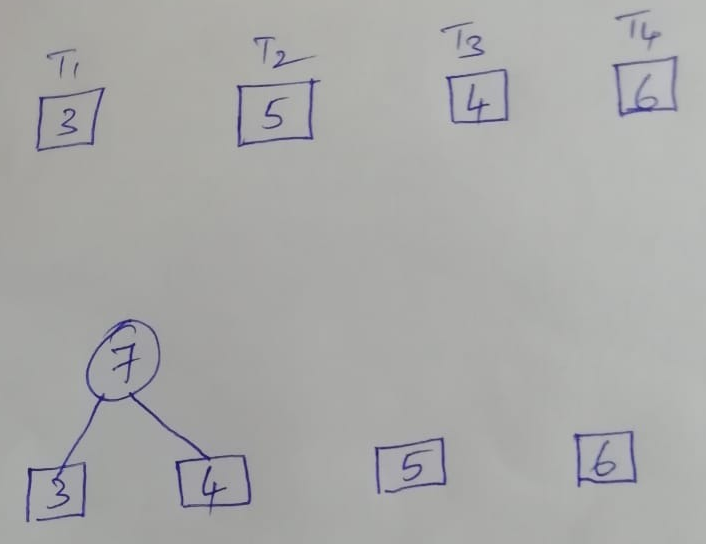
Ex: find optimal merge pattern for merging the following files

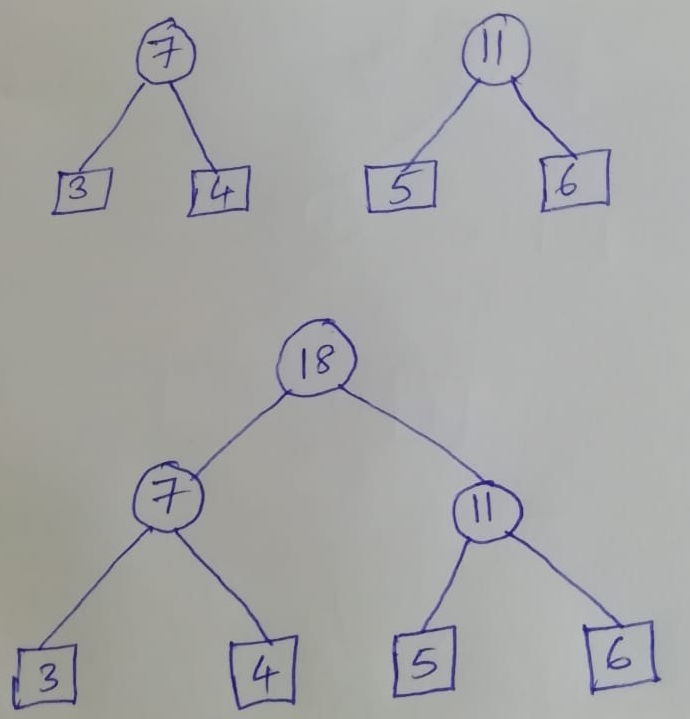
f1 with 3 values

f2 with 5 values

f3 with 4 values

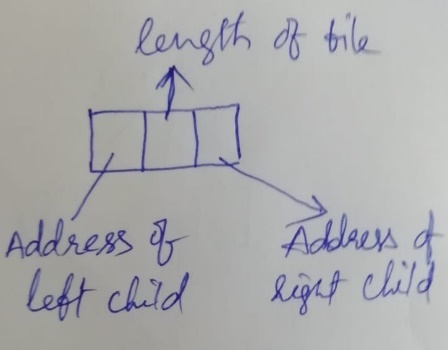
f4 with 6 values





Optimal merge pattern for merging the above four files is ((f1f3)(f2f4)).

**Algorithm of Optimal Merge Pattern problem**

Struct node 

{

node \*lptr;

int l;

node \*rptr;

};

Algorithm OMP(n, L)

// n is number of files

// L is the list containing n number of trees. Each tree contains only one node

// which represents the size of corresponding file

{

for i := 1 to n-1 do

{

ptr := new node;

ptr🡪lptr := Least(L);

ptr🡪rptr := Least(L);

ptr🡪l := ptr🡪lptr🡪l + ptr🡪rptr🡪l;

Insert(L, ptr);

}

}

**Least (L)**: selects a tree with least value at root from the list L and then deletes the selected tree from the list L.

**Insert(L, ptr)**: inserts the new tree whose root address is stored in ptr into the list L.

**Optimal Storage on Tapes problem**

Magnetic tape is a storage media which supports only sequential access. There exist n! number of orderings to store ‘n’ number of programs on the tape. For example, two orderings are possible to store two programs (1, 2) on the tape

1, 2

2, 1

Six orderings are possible to store three programs (1, 2, 3) on the tape

1, 2, 3

1, 3, 2

2, 1, 3

2, 3, 1

3, 1, 2

3, 2, 1

When the programs are stored in the order 1, 2, 3, …, n and the length of the programs are *l1,* *l2, l3,…, ln* respectively then the time needed to retrieve the program j is

If the programs are equally accessed then the total retrieval time i.e. the total time needed to retrieve all the programs is

Ex:

Let n=3 and (l1, l2, l3) = (5, 10, 3)

The possible orderings and their total retrieval times are

**Order Total Retrieval Time**

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 order with minimum retrieval time is selected as the optimal order. In above example, 3, 1, 2 is the optimal order.

*Optimal storage on tape problem* is to find an optimal order for storing the programs on the tape. The optimal order is obtained by storing the programs in increasing order of length.

*Optimal storage on tapes problem* is to find an optimal order for storing n number of programs (1, 2, 3, …, n) on m number of tapes (T0, T1, T2,…, Tm-1). To find the optimal order, at first the programs are sorted in increasing order of length. Then, the first m programs are stored on the tapes T0, T1, T2,…, Tm-1 respectively. The next m programs are stored on the tapes T0, T1, T2,…, Tm-1 respectively. This procedure is repeated until all n programs are stored.

Program i is stored on the tape Ti mod m

Ex: Find an optimal order for storing 13 programs on three tapes T0, T1, T2 where the programs are of length 12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10, 6.

1) Sort the programs in increasing order of length

10, 9, 2, 6, 13, 5, 3, 12, 11, 1, 7, 8, 4

2) Store the programs on the tapes

Store program 10 on tape T0

Store program 9 on tape T1

Store program 2 on tape T2

Store program 6 on tape T0

Store program 13 on tape T1

Store program 5 on tape T2

Store program 3 on tape T0

Store program 12 on tape T1

Store program 11 on tape T2

Store program 1 on tape T0

Store program 7 on tape T1

Store program 8 on tape T2

Store program 4 on tape T0

**Algorithm**

Algorithm OST(n, p, l, m)

// n is number of programs

// p[1:n] is an array containing the program numbers

// l[1:n] is an array containing the length of programs

// m is number of tapes

{

for i := 1 to n-1 do

{

for j := i+1 to n do

{

if l[i] > l[j] then

{

Swap(l[i], l[j]);

Swap(p[i], p[j]);

}

}

}

j := 0; // j is tape number

for i := 1 to n do

{

write(“program”, p[i], “on the tape”, j);

j := (j+1) mod m;

}

}