UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Design Analysis & Algorithm Laboratory

Greedy Approach

- Simple, straightforward and easy to implement.
- Always looking for immediate gain by overlooking the long term effect.
- Doesn't always produce the optimal solution.

Knapsack Problem

- Knapsack is similar to a container.
- Items with weights and profits.
- Some items need to be put in container in such a way total value produces a maximum profit.
- Two types of knapsack:
  - 0/1 knapsack
  - fractional knapsack
- 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.
- Items can be broken into smaller pieces in fractional knapsack.

0/1 Knapsack Problem using Greedy Approach

Capacity of the knapsack is W = 25 and the items are as shown in the following table:

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Use greedy approach to solve this problem.

0/1 Knapsack Problem using Greedy Approach

- The profit per unit weight (p_i/w_i).
- After selecting item *A*, no more item will be selected. Hence, for this given set of items total profit is *24*. Whereas, the optimal solution can be achieved by selecting items, B and C, where the total profit is *18 + 18 = 36*.
- Items cannot be broken. It should either be considered as a whole or never.

So, solution of 0/1 knapsack problem using greedy approach doesn't produce optimal solution.

Activity Selection Problem

- Greedy approach
- Perform sorting on the activities according to their finish time
- Pick the first activity from the sorted list and print
- For the remaining activities, if the start time of the activity is greater than or equal to the finish time of the previously selected activity then select the activity and print it.

Activity Selection Problem

```
i = 0;                                  // first activity always gets selected
printf("%d ", i);

for (j = 1; j < n; j++) {                       //n=total no. of activities

        if (start_time[j] >= finish_time[i])
 {

                printf ("%d ", j);
                i = j;
}
}
```

Activity Selection Problem

GREEDY- ACTIVITY SELECTOR (s, f)

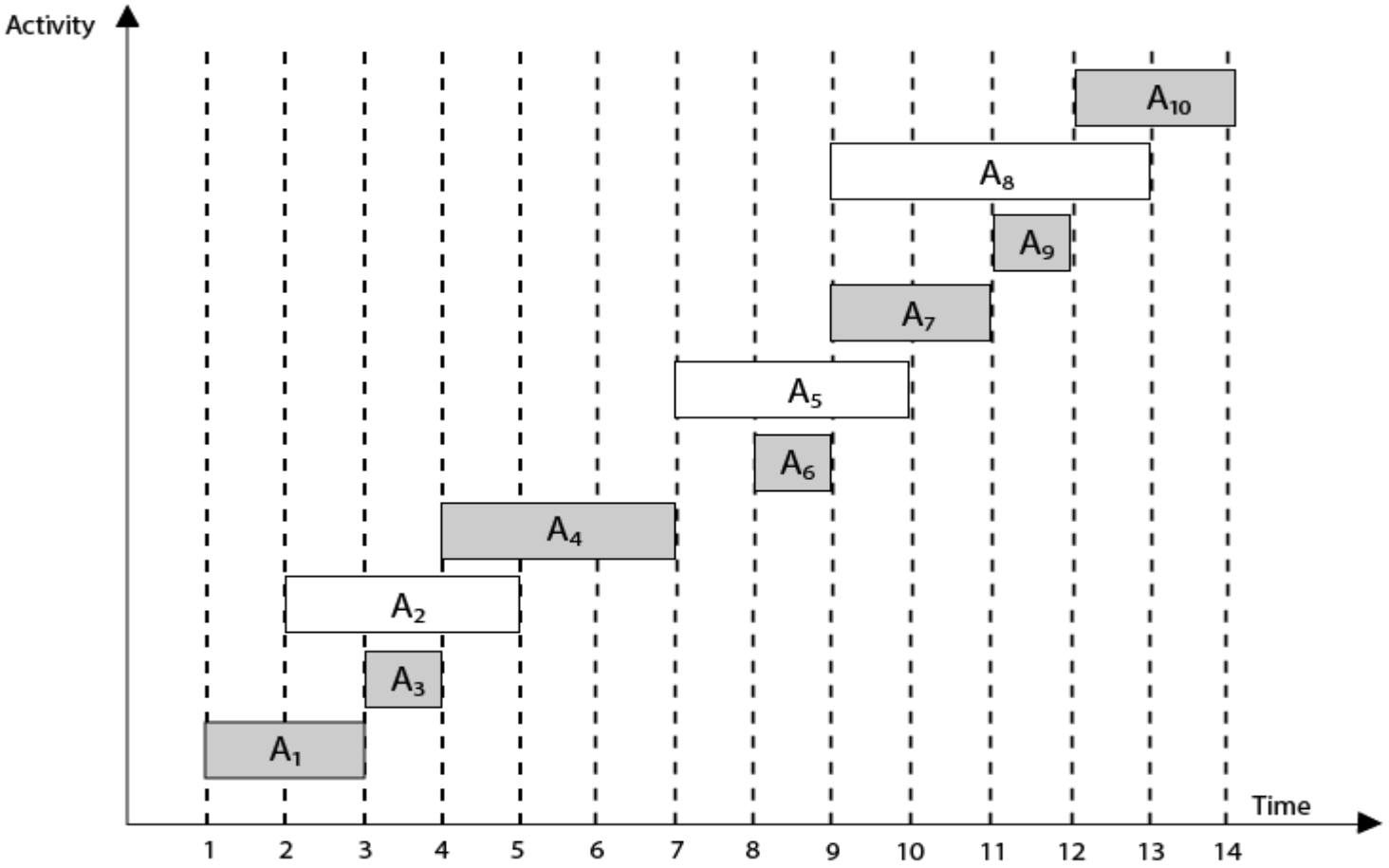
1. n ← length [s]
2. A ← {1}
3. j ← 1.
4. for i ← 2 to n
5. do if $s_i \geq f_i$
6. then A ← A ∪ {i}
7. j ← i
8. return A

Example: Given 10 activities along with their start and end time as
S = (A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A9 A10)

Si = (1,2,3,4,7,8,9,9,11,12) fi = (3,5,4,7,10,9,11,13,12,14)

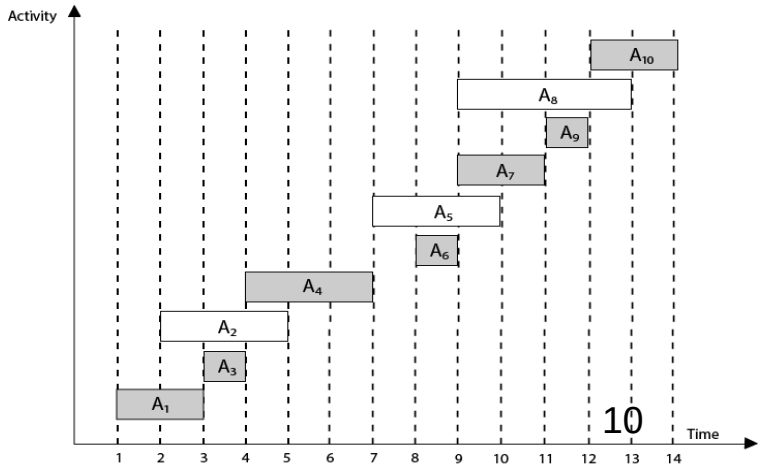
The complexity of this problem is O(n log n) when the list is not sorted. When the sorted list is provided the complexity will be O(n).

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Activity Selection Problem

Now, schedule A1
Next schedule A3 as A1 and A3 are non-interfering.
Next skip A2 as it is interfering.
Next, schedule A4 as A1 A3 and A4 are non-interfering, then next, schedule A6 as A1 A3 A4 and A6 are non-interfering.
Skip A5 as it is interfering.
Next, schedule A7 as A1 A3 A4 A6 and A7 are non-interfering.
Next, schedule A9 as A1 A3 A4 A6 A7 and A9 are non-interfering.
Skip A8 as it is interfering.
Next, schedule A10 as A1 A3 A4 A6 A7 A9 and A10 are non-interfering.
Thus the final Activity schedule is:



Optimal Merge Pattern

- Merge a set of sorted files of different length into a single sorted file where the resultant file will be generated in minimum time.

- If the number of sorted files are given, this merge can be performed pair wise which is called ***2-way merge patterns***.

Optimal Merge Pattern Example

File names	No. of elements
f1	20
f2	30
f3	10
f4	5
f5	30

Calculate the least number of comparisons needed to merge the files.

Optimal Merge Pattern Example

Merge files according to given sequence

M1 = merge f1 and f2 = 20 + 30 = 50
M2 = merge M1 and f3 = 50 + 10 = 60
M3 = merge M2 and f4 = 60 + 5 = 65
M4 = merge M3 and f5 = 65 + 30 = 95

So, the total number of comparisons needed = 50 + 60 + 65 + 95 = 270.

Now, try to optimize the comparison!

Optimal Merge Pattern Example

File names	No. of elements
f1	20
f2	30
f3	10
f4	5
f5	30

Sort the files according to the number of elements in files -

f4 , f3 , f1 , f2 , f5

Optimal Merge Pattern Example

Merge files according to ***f4 , f3 , f1 , f2 , f5*** sequence:

M1 = merge f4 and f3 = 5 + 10 = 15
M2 = merge M1 and f1 = 15 + 20 = 35
M3 = merge M2 and f2 = 35 + 30 = 65
M4 = merge M3 and f5 = 65 + 30 = 95

So, the total number of comparisons needed = 15 + 35 + 65 + 95 = 210 (<270).

Can we make it more better?

Optimal Merge Pattern

To merge a p-record file and a q-record file requires possibly p + q record moves, the obvious choice being, merge the two smallest files together at each step. Two-way merge patterns can be represented by ***binary merge trees***.

```
for i = 1 to n – 1 do
   declare new node
   node.leftchild = least (arr)
   node.rightchild = least (arr)
   node.weight) = ((node.leftchild).weight) +
            ((node.rightchild).weight)
   insert (arr, node);
return least (arr);
```

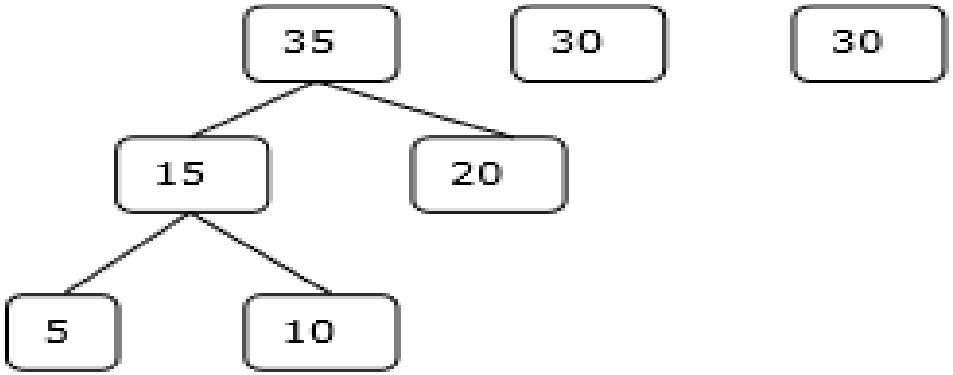
Optimal Merge Pattern Example

Set of elements

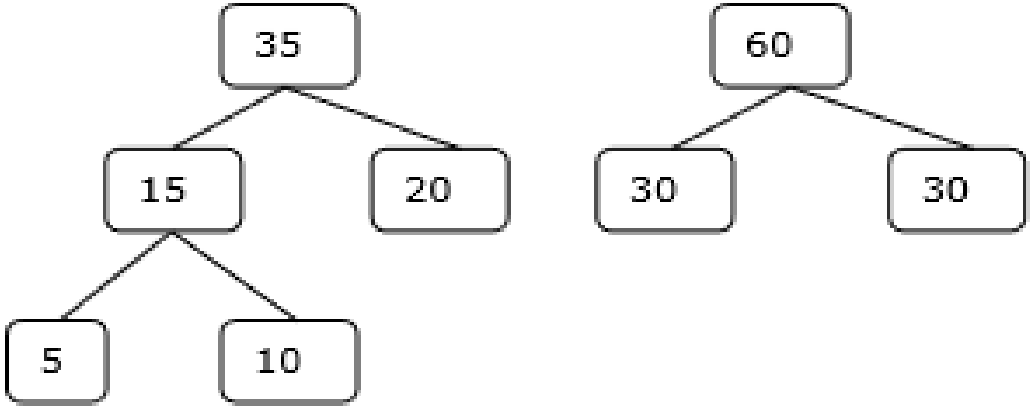
S1:

Optimal Merge Pattern Example

S2:

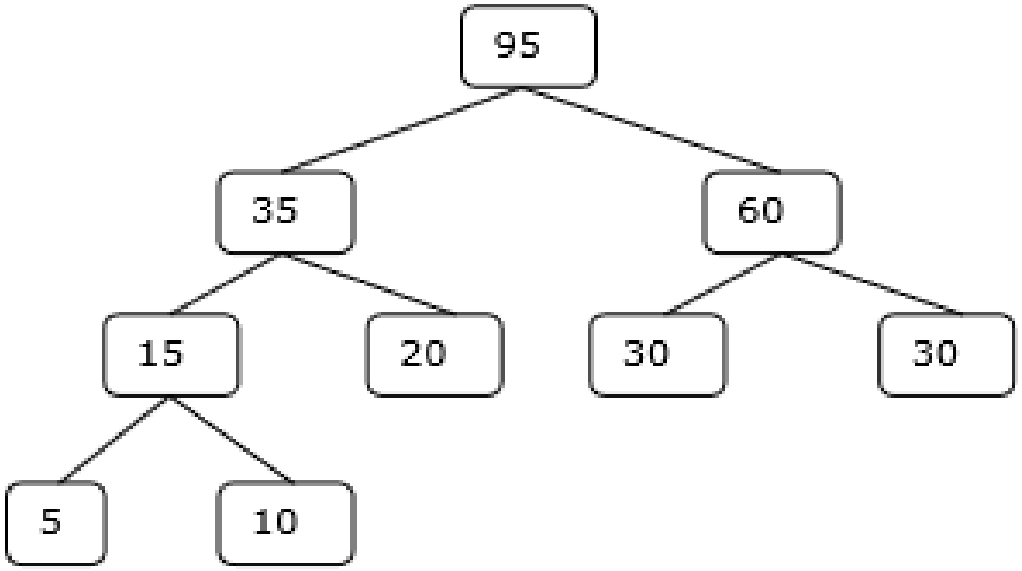


S3:



Optimal Merge Pattern Example

S4:



So, the total number of comparisons needed = 15 + 35 + 60 + 95 = 205 (<210).

Fractional Knapsack Problem

The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

This problem can be solved with the help of using two techniques:

Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.

Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.

Fractional Knapsack Problem

There are basically three approaches to solve the problem:

- The first approach is to select the item based on the maximum profit.
- The second approach is to select the item based on the minimum weight.
- **The third approach is to calculate the ratio of profit/weight.**

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
if weight + w[i] ≤ W then
 x[i] = 1
weight = weight + w[i]
else
x[i] = (W - weight) / w[i]
weight = W
break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of pi/wi, then the while loop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Fractional Knapsack Problem

Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(\frac{p_i}{w_i})$	7	10	6	5

As the provided items are not sorted based on $\frac{p_i}{w_i}$. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(\frac{p_i}{w_i})$	10	7	6	5

Fractional Knapsack Problem Example

Consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table –

First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item.

However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

Thank You