Knapsack Problem

You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

Fractional Knapsack Problem-

In Fractional Knapsack Problem,

- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

Fractional Knapsack Problem Using Greedy Method-

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01:

For each item, compute its value / weight ratio.

Step-02:

Arrange all the items in decreasing order of their value / weight ratio.

Step-03:

Start putting the items into the knapsack beginning from the item with the highest ratio.

Put as many items as you can into the knapsack.

Time Complexity-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of **Quick Sort** is O(nlogn).
- Therefore, total time taken including the sort is O(nlogn).

PRACTICE PROBLEM BASED ON FRACTIONAL KNAPSACK PROBLEM-

Problem-

For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

OR

Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

$$n = 5$$

$$w = 60 \text{ kg}$$

$$(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)$$

$$(b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)$$

OR

A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Solution-

Step-01:

Compute the value / weight ratio for each item-

Items	Weight	Value	Ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5

5	25	90	3.6

Step-02:

Sort all the items in decreasing order of their value / weight ratio-

I1	**I2**	**I5**	**I4**	**I3**
(6)	(4)	(3.6)	(3.5)	(3)

Step-03:

Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	**Items in Knapsack**	**Cost**
60	Ø	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

- Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.
- Since in fractional knapsack problem, even the fraction of any item can be taken.
- So, knapsack will contain the following items-

< I1 , I2 , I5 , (20/22) I4 >

Total cost of the knapsack

= 160 + (20/27) x 77

= 160 + 70

= 230 units

Given the weights and profits of **N** items, in the form of **{profit, weight}** put these items in a knapsack of capacity **W** to get the maximum total profit in the knapsack. In **Fractional Knapsack**, we can break items for maximizing the total value of the knapsack.

Input: *arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50*
Output: *240*
Explanation: *By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.*
Hence total price will be 60+100+(2/3)(120) = 240
Input: *arr[] = {{500, 30}}, W = 10*
Output: *166.667*

Follow the given steps to solve the problem using the above approach:

- Calculate the ratio (**profit/weight**) for each item.
- Sort all the items in decreasing order of the ratio.
- Initialize **res = 0**, curr_cap = given_cap.
- Do the following for every item **i** in the sorted order:
    - If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
    - Else add the current item as much as we can and break out of the loop.
- Return **res**.

Activity Selection Problem

The Activity Selection Problem is an optimization problem that deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. A greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?

Let's consider that you have n activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some **points to note** here:

- It might not be possible to complete all the activities since their timings can collapse.
- Two activities, say **i** and **j**, are said to be non-conflicting if si >= fj or sj >= fi where si and sj denote the starting time of activities **i** and **j** respectively, and fi and fj refer to the finishing time of the activities **i** and **j** respectively.
- **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

- act[] array containing all the activities.
- s[] array containing the starting time of all the activities.
- f[] array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- sol[] array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array act[] and add it to sol[] array.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

Step 5: Select the next activity in act[] array.

Step 6: Print the sol[] array.

Activity Selection Problem Example

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

A possible **solution** would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}**.

Step 3: Repeat the steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]

For the data given in the above table,

A. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. s(a3) > f(a2)), we add **a3** to the solution set. Thus **sol = {a2, a3}**.

B. Select **a4**. Since s(a4) < f(a3), it is not added to the solution set.

C. Select **a5**. Since s(a5) > f(a3), **a5** gets added to solution set. Thus **sol = {a2, a3, a5}**

D. Select **a1**. Since s(a1) < f(a5), **a1** is not added to the solution set.

E. Select **a6**. **a6** is added to the solution set since s(a6) > f(a5). Thus **sol = {a2, a3, a5, a6}**.

Step 6: At last, print the array sol[]

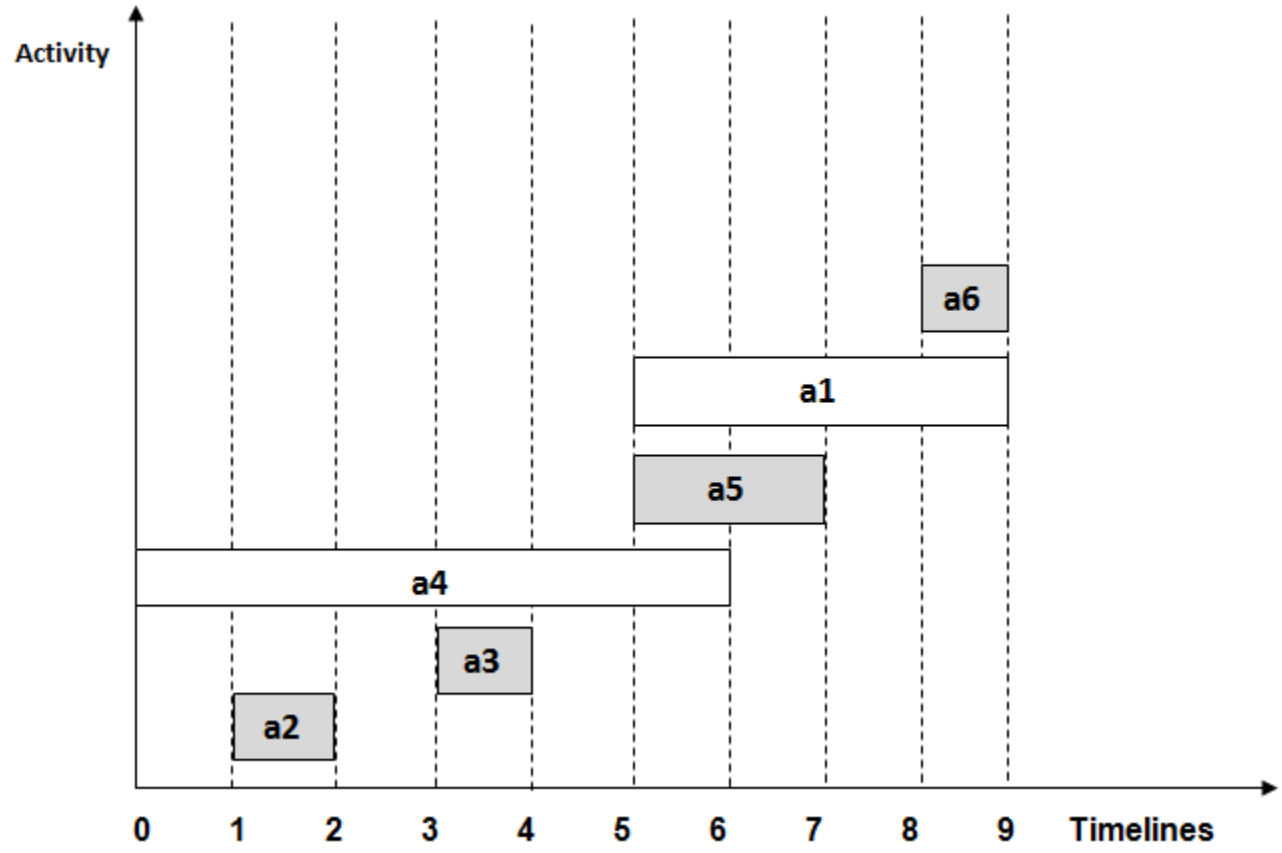
Hence, the execution schedule of maximum number of non-conflicting activities will be:

```
(1,2)
```

(3,4)

(5,7)

(8,9)



In the above diagram, the selected activities have been highlighted in grey.

Implementation of Activity Selection Problem Algorithm

Now that we have an overall understanding of the activity selection problem as we have already discussed the algorithm and its working details with the help of an example, following is the C++ implementation for the same.

Note: The algorithm can be easily written in any programming language.

```
#include <bits/stdc++.h>
using namespace std;
#define N 6                    // defines the number of activities
// Structure represents an activity having start time and finish time.
struct Activity
{
   int start, finish;
};
// This function is used for sorting activities according to finish time
bool Sort_activity(Activity s1, Activity s2)
{
   return (s1.finish< s2.finish);
}
/*          Prints maximum number of activities that can
            be done by a single person or single machine at a time.
*/
void print_Max_Activities(Activity arr[], int n)
{
   // Sort activities according to finish time
            sort(arr, arr+n, Sort_activity);


            cout<< "Following activities are selected \n";
   // Select the first activity
   int i = 0;
            cout<< "(" <<arr[i].start<< ", " <<arr[i].finish << ")\n";
   // Consider the remaining activities from 1 to n-1
   for (int j = 1; j < n; j++)
   {
            // Select this activity if it has start time greater than or equal
            // to the finish time of previously selected activity
            if (arr[j].start>= arr[i].finish)
```

```
            {
                        cout<< "(" <<arr[j].start<< ", "<<arr[j].finish << ") \n";
                        i = j;
            }
    }
}
// Driver program
int main()
{
    Activity arr[N];
            for(int i=0; i<=N-1; i++)
            {
                    cout<<"Enter the start and end time of "<<i+1<<" activity \n";
                    cin>>arr[i].start>>arr[i].finish;
    }

            print_Max_Activities(arr, N);
    return 0;
}
```

Output

```
Enter the start and end time of 1 activity
5 9
Enter the start and end time of 2 activity
1 2
Enter the start and end time of 3 activity
3 4
Enter the start and end time of 4 activity
0 6
Enter the start and end time of 5 activity
5 7
Enter the start and end time of 6 activity
8 9
Following activities are selected
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

Time Complexity Analysis

Following are the scenarios for computing the time complexity of Activity Selection Algorithm:

- **Case 1**: When a given set of activities are already sorted according to their finishing time, then there is no sorting mechanism involved, in such a case the complexity of the algorithm will be O(n)
- **Case 2**: When a given set of activities is unsorted, then we will have to use the sort() method defined in **bits/stdc++** header file for sorting the activities list. The time complexity of this method will be O(nlogn), which also defines complexity of the algorithm.

Real-life Applications of Activity Selection Problem

Following are some of the real-life applications of this problem:

- Scheduling multiple competing events in a room, such that each event has its own start and end time.
- Scheduling manufacturing of multiple products on the same machine, such that each product has its own production timelines.
- Activity Selection is one of the most well-known generic problems used in Operations Research for dealing with real-life business problems.
