1. **Activity Selection Problem**

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

**Example:**

**Example 1 :** Consider the following 3 activities sorted by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most **two** activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[] ]

**Example 2 :** Consider the following 6 activities sorted by by finish time.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

A person can perform at most **four** activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[] ]

**Solution:**

1. Sort the activities according to their finishing time
2. Select the first activity from the sorted array and print it.
3. Do following for remaining activities in the sorted array.  
   .a) If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

void printMaxActivities(int s[], int f[], int n)

{

    int i, j;

    printf ("Following activities are selected n");

    // The first activity always gets selected

    i = 0;

    printf("%d ", i);

    // Consider rest of the activities

    for (j = 1; j < n; j++)

    {

      /\* If this activity has start time greater than or equal to the finish time of previously selected activity, then select it \*/

      if (s[j] >= f[i])

      {

          printf ("%d ", j);

          i = j;

      }

    }

}

1. **Task scheduling/ Job sequencing problem**

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

struct Job

{char id;      // Job Id

   int dead;    // Deadline of job

   int profit;  // Profit if job is over before or on deadline

};

bool comparison(Job a, Job b)

{

     return (a.profit > b.profit);

}

void printJobScheduling(Job arr[], int n)

{

    sort(arr, arr+n, comparison); //using STL

    int result[n]; // To store result (Sequence of jobs)

    bool slot[n];  // To keep track of free time slots

    // Initialize all slots to be free

    for (int i=0; i<n; i++)

        slot[i] = false;

    // Iterate through all given jobs

    for (int i=0; i<n; i++)

    {

       // Find a free slot for this job (Note that we start from the last possible slot)

       for (int j=min(n, arr[i].dead)-1; j>=0; j--)

       {

          // Free slot found

          if (slot[j]==false)

          {

             result[j] = i;  // Add this job to result

             slot[j] = true; // Make this slot occupied

             break;

          }

       }

    }

    // Print the result

    for (int i=0; i<n; i++)

       if (slot[i])

         cout << arr[result[i]].id << " ";

}

1. **Fractional Knapsack Problem**

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Knapsack Capacity, W = 50;

Output:

Maximum possible value = 240

By taking full items of 10 kg, 20 kg and 2/3rd of last item of 30 kg;

struct Item

{

    int value, weight;

 };

bool cmp(struct Item a, struct Item b)

{

    double r1 = (double)a.value / a.weight;

    double r2 = (double)b.value / b.weight;

    return r1 > r2;

}

double fractionalKnapsack(int W, struct Item arr[], int n)

{

    sort(arr, arr + n, cmp);   //sorting Item on basis of ratio

     int curWeight = 0;  // Current weight in knapsack

    double finalvalue = 0.0; // Result (value in Knapsack)

    // Looping through all Items

    for (int i = 0; i < n; i++)

    {

        // If adding Item won't overflow, add it completely

        if (curWeight + arr[i].weight <= W)

        {

            curWeight += arr[i].weight;

            finalvalue += arr[i].value;

        }

        // If we can't add current Item, add fractional part of it

        else

        {

            int remain = W - curWeight;

            finalvalue += arr[i].value \* ((double) remain / arr[i].weight);

            break;

        }

    }

    // Returning final value

    return finalvalue;

}

Few more Greedy algorithms to be discussed in last session…..

Prim’s

Kruskal’s

Dijkstra etc