Weighted Job Scheduling

Given N jobs where every job is represented by following three elements of it.

1. Start Time
2. Finish Time
3. Profit or Value Associated (>= 0)

Find the maximum profit subset of jobs such that no two jobs in the subset overlap.

Example:

```
Input: Number of Jobs n = 4

       Job Details {Start Time, Finish Time, Profit}

       Job 1:  {1, 2, 50}

       Job 2:  {3, 5, 20}

       Job 3:  {6, 19, 100}

       Job 4:  {2, 100, 200}

Output: The maximum profit is 250.

We can get the maximum profit by scheduling jobs 1 and 4.

Note that there is longer schedules possible Jobs 1, 2 and 3

but the profit with this schedule is 20+50+100 which is less than
250.
```

The above problem can be solved using the following recursive solution.

```
1) First sort jobs according to finish time.
2) Now apply following recursive process.
   // Here arr[] is array of n jobs
   findMaximumProfit(arr[], n)
   {
     a) if (n == 1) return arr[0];
     b) Return the maximum of following two profits.
         (i) Maximum profit by excluding current job, i.e.,
             findMaximumProfit(arr, n-1)
         (ii) Maximum profit by including the current job
   }
```

How to find the profit including current job?

```
The idea is to find the latest job before the current job (in
sorted array) that doesn't conflict with current job 'arr[n-1]'.
Once we find such a job, we recur for all jobs till that job and
```

```
add profit of current job to result.
In the above example, "job 1" is the latest non-conflicting
for "job 4" and "job 2" is the latest non-conflicting for "job 3".
```

The following is the implementation of the above naive recursive method.

C++

```cpp
// C++ program for weighted job scheduling using Naive Recursive Method
#include <iostream>
#include <algorithm>
using namespace std;

// A job has start time, finish time and profit.
struct Job
{
    int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool jobComparator(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]. If there is no compatible job,
// then it returns -1.
int latestNonConflict(Job arr[], int i)
{
    for (int j=i-1; j>=0; j--)
```

```
    {
        if (arr[j].finish <= arr[i-1].start)
            return j;
    }
    return -1;
}

// A recursive function that returns the maximum possible
// profit from given array of jobs.  The array of jobs must
// be sorted according to finish time.
int findMaxProfitRec(Job arr[], int n)
{
    // Base case
    if (n == 1) return arr[n-1].profit;

    // Find profit when current job is included
    int inclProf = arr[n-1].profit;
    int i = latestNonConflict(arr, n);
    if (i != -1)
      inclProf += findMaxProfitRec(arr, i+1);

    // Find profit when current job is excluded
    int exclProf = findMaxProfitRec(arr, n-1);

    return max(inclProf,  exclProf);
}
```

```
// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, jobComparator);

    return findMaxProfitRec(arr, n);
}

// Driver program
int main()
{
    Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "The optimal profit is " << findMaxProfit(arr, n);
    return 0;
}
```

Output:

```
The optimal profit is 250
```

The above solution may contain many overlapping subproblems. For example, if lastNonConflicting() always returns the previous job, then findMaxProfitRec(arr, n-1) is called twice and the time complexity becomes $O(n*2^n)$. As another example when lastNonConflicting() returns previous to the previous job, there are two recursive calls, for n-2 and n-1. In this example case, recursion becomes the same as Fibonacci Numbers.
So this problem has both properties of Dynamic Programming, Optimal Substructure, and Overlapping Subproblems.

Like other Dynamic Programming Problems, we can solve this problem by making a table that stores solutions of subproblems.
Below is an implementation based on Dynamic Programming.

C++

```
// C++ program for weighted job scheduling using Dynamic
// Programming.
#include <algorithm>
#include <iostream>
using namespace std;

// A job has start time, finish time and profit.
struct Job {
    int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool jobComparator(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]
int latestNonConflict(Job arr[], int i)
{
    for (int j = i - 1; j >= 0; j--) {
        if (arr[j].finish <= arr[i].start)
```

```
            return j;
    }
    return -1;
}

// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr + n, jobComparator);

    // Create an array to store solutions of subproblems.
    // table[i] stores the profit for jobs till arr[i]
    // (including arr[i])
    int* table = new int[n];
    table[0] = arr[0].profit;

    // Fill entries in M[] using recursive property
    for (int i = 1; i < n; i++) {
        // Find profit including the current job
        int inclProf = arr[i].profit;
        int l = latestNonConflict(arr, i);
        if (l != -1)
            inclProf += table[l];

        // Store maximum of including and excluding
```

```
        table[i] = max(inclProf, table[i - 1]);
    }

    // Store result and free dynamic memory allocated for
    // table[]
    int result = table[n - 1];
    delete[] table;

    return result;
}

// Driver program
int main()
{
    Job arr[] = { { 3, 10, 20 },
                  { 1, 2, 50 },
                  { 6, 19, 100 },
                  { 2, 100, 200 } };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The optimal profit is "
         << findMaxProfit(arr, n);
    return 0;
}
```

```
The optimal profit is 250
```

Time Complexity of the above Dynamic Programming Solution is $O(n^2)$. Note that the above solution can be optimized to O(nLogn) using Binary Search in latestNonConflict() instead of linear search. Thanks to Garvit for suggesting this optimization. Please refer below post for details.