To: Professor Kearns
From: Anh Nguyen
Class: CPE 349 - 09
Date: 11/5/2015
Assignment 4 : Weighted Interval Scheduling

Overview:

Practice solving dynamic programming problem

Analysing problem:

Goal: Maximize total value with the constraint that tasks must not overlapping each other.
Problem size: Number of tasks
Assume : that the list of tasks are already sort by order of ascending finish time. (if the list aren't sorted yet, use merge sort or quicksort to sort it)

Solution:

1. **The recurrence relation**

   At any task in the list of task, we have two option, either to do this task or not.

   - If we do this task, the maximum total value will be the value of this task + value of all compatible previous tasks.
   - If we choose not to do this task, the maximum total value will remain as it was before coming to this task.

   From those relations, formulate the recurrence relation:

   $$\mathbf{F(n) = \max\left(v[n] + F[prev(n)] \ , \ F[n-1]\right)}$$

   where: F(n) : maximum total value of tasks that not overlapping each other up till task n
   v[n] : value of singular task at index n
   prev(n) : index of the previous task which is compatible with task n
   F[n-1] : maximum total value of tasks that not overlapping each other up till task n-1

   Base case is when there is no task, therefore, the maximum total value is 0

2. **Specification of table:**

   For this list of tasks (sorted list) : (start , end) value

   (2 , 6) 5
   (4 , 7) 6
   (7 , 9) 8
   (3 , 10) 6
   (10,13) 2

   I construct a 2-dimensional table below

Interval	basecase	(2,6)	(4,7)	(7,9)	(3,10)	(10,13)
Value						
Prev(i)						
F(i)						

Interval: The first row are the task intervals sorted in ascending finish time order.
Value: The value of each task interval
Prev(i): The index of the previous task which is compatible (finish before this task start)
F(i): The maximum total value of tasks which are compatible.

3. The specification of the algorithm.

Starting from the bottom of this problem (base-case), the next larger problem is when there is 1 task. The max total value of 1 task is clearly that task's value. The next larger is 2 tasks. Now we have to check if the first task was overlapping with second task or not. If they were not overlapping, then the max total value is the sum of first and second value. If they were overlapping, then we have to choose to take 2nd task or not. The decision is made base on the recurrence relation formula above. Below is how to apply this bottom-up algorithm to the table.

- Before filling in the table, know that the base case is when there is no task (Ø), so there is no previous task, and the *value = 0*.
- Fill in the row *value* as given.
- To compute the *Prev(i)* , look for the last previous task which is compatible with this task. Fill in the *Prev(i)* entry with the index of that previous compatible task just found. If there is no previous compatible task, *Prev(i) = 0* (basecase).
- To compute *F(i)* , use the recurrence relation above. That mean, we have to compute 2 things below, and take the max of it:
    - v[n] + F[prev(n)] : Value of this task + the max total value of the previous compatible task
    - F[n-1]: max total value of the task right before this task (not need to be compatible with this task)

Interval	basecase Ø	(2,6)	(4,7)	(7,9)	(3,10)	(10,13)
Value	0	5	6	8	6	2
Prev(i)	0	0	0	2	0	4
F(i)	0	5	6	13	6	15

4. Derivation of the closed form solution.

- We had this recurrence relation:

  $F(n) = \max (v[n] + F[prev(n)] , F[n-1])$

- For each computation of this recurrence, we have to search for prev(n), this process has runtime O(logn)
- After finding prev(n), the worse case occur when prev(n) is (n-1)

  $\Rightarrow F(n) = v[n] + F(n-1)$

  In term of running time, $v[n] = O(1)$

  $\Rightarrow F(n) = F(n-1) + 1$

  Using back substitution:

  $F(n) = F(n-1) + 1$
  $= (F(n-2) +1) + 1$
  $= F(n-2) +2$
  $= (F(n-3) +1) + 2$
  $= F(n-3) +3$

  $\Rightarrow F(n) = F(n-k) + k$

  Substitute (k = n), we have

  $F(n) = F(n-n) + n$

  $\Rightarrow F(n) = F(0) + n$

  $\Rightarrow F(n) = O(n)$

  Combine with searching for prev(n) , the total runtime is:

  $O(n) * O(logn) = O(nlogn)$