Michael Kaiser
Assignment 2
Comp 510
Fall 2012

16.1-1
Give a dynamic-programming algorithm for the activity-selection problem, based on the recurrence (16.2). Have your algorithm compute the sizes c[i, j] as defined above and also produce the maximum-size subset A of activities. Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY- ACTIVITY-SELECTOR.

Let the recurrence relationship be:

$c[i,j] = \{ 0 \quad \text{if } S_{ij} = 0 \}$
$\{\max(c[i,k] + c[k,j] + 1) \quad \text{if } S_{ij} \neq 0 \}$

S_{ij} is a set containing activities from the full set S between i and j. The full solution is when i = 0 and j = n + 1 (where n is the number of activities).

```
DYNAMIC-ACTIVITY-SELECTON(s, f)
{
        c = {0};                //Set table to 0
        k_max_table = {-1}; //Flag as empty
        for(x = 1 to n)
        {
                for(i = 0 to n – x + 1)
                {
                        j = i+x;
                        k_max_table[i,j] = -1;
                        if(f[i] < s[j])    //There is room for the activity
                        {
                                for(k=i+1 to j – 1)
                                {
                                        if(s[k] >= f[i] && f[k] <= s[j]) //if k is compatible
                                        {
                                                kval =  c[i,k] + c[k,j] + 1;
                                                if(kval > c[i,j])
                                                {
                                                        c[i,j] = kval;
                                                        k_max_table[i,j] = k;
                                                }
                                        }

                                }
                        }
                }
        }
}
```

Each cell takes is computed within three nested for loops. Considering this, it is easy to come to the conclusion that the algorithm takes $O(n^3)$ time, which is less efficient than greedy algorithm's O(n) time.

16.1-2
Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

This approach is the exact same approach the book uses for the greedy algorithm, however it is performing the algorithm from the opposite direction of the set. From this, we can conclude it has the same running time, is greedy, and is also therefore another optimal solution.

16.1-3
Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

Least duration

[i]	1	2	3
s_i	0	3	5
f_i	5	6	10

From this table, we would select only A_2, which is incorrect because a_1,a_3 is the optimal solution.

Overlaps the fewest

[i]	1	2	3	4	5	6	7	8	9	10	11
s_i	0	1	1	1	2	3	4	5	5	5	6
f_i	2	3	3	3	4	5	6	7	7	7	8
ovlap	3	4	4	4	4	2	4	4	4	4	3

Using the algorithm described, we are forced to choose a_6. The optimal solution is a_1,a_5,a_7,a_{11}, which does not contain a_6. Thus the algorithm cannot choose the optimal solution.

Earliest start time

[i]	1	2	3
s_i	0	2	4
f_i	20	4	8

The maximum set is obviously a_2,a_3 here, but the earliest time is a_1, and would be chosen by this greedy algorithm.

16.2-1
Prove that the fractional knapsack problem has the greedy-choice property.

The greedy-choice algorithm for the fractional knapsack problem involves the thief grabbing the highest value per weight object in the knapsack until the supply of that item is exhausted, and then begins grabbing the second highest value per weight, and so on. Due to the structure of this problem, there is only one sub-problem to solve (which is the next highest value per weight item), and is greedy by definition (what is the most valuable item per weight currently available at this moment in time). So by definition, this has the greedy-choice property because the greedy algorithm generates the optimal solution.

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in O(n W) time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Define c[i,w] to be the value of the solution for items 1,...,i and maximum weight w such that

$$c[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0, \\ c[i-1,w] & \text{if } w_i > w, \\ \max(v_i + c[i-1,w-w_i], c[i-1,w]) & \text{if } i>0 \text{ and } w \geq w_i. \end{cases}$$

Where v_i is the value of item i, and w_i is the weight of item i. The following algorithm will fill the table c such that c[n,W] contains the maximum value that the thief can take:

```
DYNAMIC-KNAPSACK(v,w,n,W)
table c = {0};
for(i=1 to n)
{
        for(w=1 to W)
        {
                if(wi<=w)       //Weight of item is less than remaining weight
                {
                        if(vi + c[i-1,w-wi] > c[i-1,w])          //max(vi + c[i-1,w-wi], c[i-1,w])
                        {
                                c[i,w] = vi + c[i-1,w-wi];
                        }
                        else
                        {
                                c[i,w] = c[i-1,w];
                        }
                }
                else
                {
                        c[i,w] = c[i-1,w];
                }
        }
}
```

The algorithm above takes O(nW) time, as the nested for loops dictate this. The table c can be traced to provide the solution from the bottom of the table to c[n,W].

16.5-1

Solve the instance of the scheduling problem given in Figure 16.7, but with each penalty w_i replaced by $80 - w_i$

The figure for this problem will be as follows:

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	10	20	30	40	50	60	70

Just looking at the chart it is fairly straight forward: a_5 has a high penalty at 1, so it must go first. The five other tasks (excluding a_7) need to fit within the 4th deadline, so we choose the three with the highest penalty to not incur the penalty (a_6, a_4 and a_3). The two which will inevitably incur the penalty are therefore a_1 and a_2, so we schedule a_7, followed by the two with penalties and we've got the optimal solution.

Therefore, we have:
$\{a_5,a_6,a_4,a_3,a_7,a_2,a_1\} = 30$

16-2

Suppose you are given a set S = {a1, a2,....an} of tasks, where task ai requires pi units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let ci be the completion time of task ai , that is, the time at which task ai completes processing. Your goal is to minimize the average completion time, that is, to minimize (average).

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

The algorithm that solves this problem is quicksort (or any efficient sort), so the efficiency is $O(n\log(n))$. The reason why this works is because a greedy algorithm will solve this problem, because the average will be minimized if the shortest task runs first. This is because the average can be written as:

average = $(p_1 + (p_1 + p2) + ... + (p_1 + ... + p_n)) / n$.

The later the longer running programs are ran, the fewer times it will be added to the sum, thus minimizing the average, and thus sorting solves this problem.

b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its release time r_i. Suppose also that we allow preemption, so that a task can be suspended and restarted at a later time. Give an algorithm that schedules the tasks so as to minimize that average completion time in this new scenario. Prove that your algorithm minimizes that average completion time, and state the running time of your algorithm.

Under the same principle that part a was solved, part b will be solved by using the Shortest Remaining Time First (SRTF) algorithm for CPU scheduling. Essentially we take the total time our CPU is computing, and for each unit of time, we look at all available tasks and choose the shortest remaining time left task for the one unit of time of execution.

Thus, for analyzing the running time of the algorithm, let us assume the CPU will never be

idle (no gaps in task availability), we have n tasks, and the sum of the execution of all tasks is t. The algorithm is as follows

```
SRTF(a,p,r,t)
{
	for(i=0 to T)
	{
		min = ∞;
		index_of_min = -1;
		for(j=0 to n)
		{
			if((i > r[j]) && (min > p[j]) && (p[j] != 0) //If over release time, and SRT
			{
				min = p[j];
				index_of_min = j;
			}
		}
		p[index_of_min] = p[index_of_min] – 1;
		if(p[index_of_min] == 0)
		{
			sum = sum + i;
		}
	}
	avg = sum / n;
}
```

The running time of this algorithm is therefore O(nT). This will generate the optimal solution because the running sum, as discussed in the last part, will have smaller values if we finish the shortest remaining time first tasks.

References

Class Textbook and solution manual (3rd ED)
http://www.eng-hs.net/files/algorithms-solutions-ch-16-oct2010.pdf