Homework #3

due Friday, 16 March 2018, at 11:45 PM

[Total points: 28]

1. [9 points.] ***Applying dynamic programming to a new problem***

Suppose you have inherited the rights to 500 previously unreleased songs recorded by the popular group Raucous Rockers. You plan to release a set of 5 CD's (numbered 1 through 5) with a selection of these songs. Each disk holds a maximum of 60 minutes of music and each song must appear in its entirety on one disk. Since you are a classical music fan and have no way of judging the artistic merits of the songs, you decide to use the following criteria: the songs will be recorded in order by the date they were written, and the number of songs included will be maximized. Suppose you have a list $l_1, l_2, ..., l_{500}$ of the lengths of the songs, in order by the date they were written (no song is more than 60 minutes long).

Give an algorithm to determine the maximum number of songs that can be included using the given criteria. Hint: Let $T[i][j]$ be the minimum amount of time needed for recording any i songs from among the first j songs. You should interpret T to include blank time at the end of a completed disk. For example, if a selection of songs uses all of the first disk plus 15 minutes of the second disk, the time for that selection should be 75 minutes even if there is blank space at the end of the first disk. Generalize your solution to a situation where there are m CD's, n songs, and a capacity c for each of the CD's. What is the asymptotic runtime in terms of m and n [1 point]?

In order to get full credit, you need to give a recursive formula [4 points] and your algorithm needs to retrieve the list of CD's [2 points].

Sol.

Let *T* be a 2D array of size *n x n* (where *n* = number of songs, and *n = 500* according to the question) where *T[i][j]* denotes the minimum amount of time needed to record any i songs from the first j songs. Therefore, *T[i][j]* should be maximum such that *T[i][j]* ≤ c*m (where c = capacity of each CD and m = number of CDs)

Let free_space be the total amount of free space on the current CD after choosing (i-1) songs among the first (j-1) songs.

Recursive Formula:

There are two cases to populate T[i][j],

Case 1: i > j : T[i][j] = 0

Case 2 : i ≤ j :

if(free_space < l_j):

then T[i][j] = min(T[i-1][j-1] + l_j+free_space, T[i][j-1])

else :

T[i][j] = min(T[i-1][j-1]+l_j, T[i][j-1])

Pseudo – Code:

SOLVE-RAUCOUS-ROCKERS(m,c,n):

```
N_MAX=0, TIME_MAX=0, T[i][j]=0, visited[i][j]=0 ∀ (1 ≤ i,j ≤ n)
for i in 1 to n:
      for j in 1 to n:
            free_space = c*(1 + floor(T[i−1][j−1]/c)) - T[i-1][j-1]
```

```
            if i>j:
                  T[i][j] = 0
            else:
                  if free_space <= l_j:
                  T[i][j] = min(T[i-1][j-1] + l_j + free_space, T[i][j-1])
                        if T[i-1][j-1] + l_j + free_space < T[i][j-1]:
                              visited[i][j] = TRUE
                  else:
                        T[i][j] = min(T[i-1][j-1]+l_j, T[i][j-1])
                        if T[i-1][j-1] + l_j  < T[i][j-1]:
                              visited[i][j] = TRUE
                  endif
                  if T[i][j] ≤ c * m AND i > N_MAX:
                        N_MAX = i
                        TIME_MAX = T[i][j]
                  endif
      endfor
endfor
```

To fetch the list of CD's, we need the visited array which we have populated in the previous function,

FETCH_CDS(visited, n):

```
Inititalize empty List L
for i from n down to 1:
      for j from n down to 1:
            if visited[i][j] == TRUE:
                  L.append(l_j)
            endif
      endfor
endfor
```

Time Complexity: $O(n^2)$

2. [12 points, 2 each for (a) and (b), 4 each for (c) and (d).] *Applying dynamic programming to a new problem*

Recall the activity selection problem of Section 16.1 (also the first greedy algorithm example presented in class). Suppose that, in addition to start and finish times, each activity a_i also has a weight w_i and that the goal is to maximize the total weight of the selected activities.

(a) Prove that the greedy strategy used for the unweighted version fails to give the optimum solution for the weighted version of the problem.

(b) Prove that a greedy strategy that repeatedly selects the activity with largest weight among the remaining activities fails to give the optimum solution.

(c) Give a recursive formulation for the optimal solution to the weighted problem. Don't forget the base case.

(d) Give pseudocode for an algorithm that not only finds the weight of the optimum solution but also returns a list of the activities selected. Analyze the runtime of your algorithm. For full credit your algorithm should be linear except for an O($n$lg$n$) preprocessing step [This part is worth 1 point each for the formula and the algorithm]. You may have to rethink your recursive formula to get this bound.

Sol.

(a) Let us take an example for this case,

i	1	2	3	4	5
s_i	1	2	3	4	5
f_i	2	4	10	8	12
w_i	10	20	10	5	100

*[*where s_i denotes the start time of the activity i, f_i denotes the finish time of the activity i, and w_i is the weight associated with activity i*]*

Using Greedy Approach on this weighted activity set, the activities selection will be like following,

Finish time is lowest from activity 1, so it is selected, then the next is activity 2 because its start time is just after the finish time on activity 1, next activity 4 is selected using the same rule. So, the total weight of the selected activities is, $w_1 + w_2 + w_4 = 10 + 20 + 5 = \mathbf{35}$

This is clearly not the best selection, as one of the activity selections could be 1, 2 and 5. The total weight of this selection is, $w_1 + w_2 + w_5 = 10 + 20 + 100 = \mathbf{130}$

So, this disproves the applicability of unweighted greedy approach on the weighted activity problem.

(b) Let us take an example for this case,

i	1	2	3	4	5	6	7
s_i	1	1	10	20	30	40	90
f_i	100	10	20	30	40	90	100
w_i	100	20	20	20	20	20	20

Using the greedy approach selecting activity with largest weight among remaining activities, the selection of activities will be like,

First activity to be selected will be activity 1, as it has highest weight. There is no other activity that can be selected after this selection, so the total weight of selected activities is w_1 = ***100***.

On the other hand, if activities selected are w_2, w_3, w_4, w_5, w_6, w_7 then the total weight of selected activities is 20 + 20 + 20 + 20 + 20 + 20 = ***120***

So, this disproves the applicability of the given approach on the weighted activity selection problem.

(c) RECURSIVE FORMULATION OF THE OPTIMAL SOLUTION

Let s_i denote the start time of the activity *i*, f_i denote the finish time of the activity *i*, and w_i is the weight associated with activity *i.* Also let *s,f, and w* contain list of values corresponding to start time, finish time and weights respectively for activities.

And let *W(k)* denote the maximum possible weight for scheduling activities using first *k* activities and *prev(i)* denote the last activity which ends before beginning of activity a_i.

Using the above notation, we can formulate a recursive solution to the weighted problem as,

$$W(k) = 0 \quad \text{, if } k = 0 \text{ (i.e., no activities to be scheduled)}$$
$$= \max\{W(prev(k)) + w_k, W(k-1)\} \text{ , otherwise}$$

Proof of correctness of this recursion:

The base case is when no activity is scheduled, and hence the value of weight should be set to 0 in that scenario.

For an activity a_k, there are two possible cases while handling activity selection problem, i.e., either to include the activity a_k, or exclude the activity a_k.

Case 1: *If activity* a_k *is not present in the set of scheduled activities*

In this case, it is evident that activities that are part of the scheduled activities, are from the first *(k-1)* activities. So, it can be concluded that value of sum of all *(k-1)* activities will never be greater than the value of *W(k-1)*.

So, *Sum of (k-1) activities* ≤ *W(k-1)* ***.... Eqn. (1)***

On the other hand, W(k-1) should not exceed the sum of first (k-1) activities, as this would contradict the optimality property of the solution, i.e., if this is assumed to be true, we would then be able to find an activity which would increase the value of *W(k-1)*, which is not true.

So, *Sum of (k-1) activities* ≥ *W(k-1)* ***.... Eqn. (2)***

From Eqn. (1) and Eqn. (2), we can conclude that the optimal value of *W(k)* when a_k is not included will be optimal value of including *(k-1)* activities, i.e., ***W(k-1)***

Case 2: *If activity* a_k *is present in the set of scheduled activities*

If a_k is included in the set of scheduled activities, then it is not possible to include any other activity a_i, that lies after *prev[i]* and before a_k, as it would then overlap with existing activities. So, it can be concluded for all activities in the set of scheduled activities, before a_k will follow the following inequality,

Sum of activities scheduled till prev[k] ≤ *W(prev[k])*

As by the property of optimality, we can clearly conclude that sum of activities scheduled till activity *prev[k]* should be at-max *W(prev[k])*, otherwise *W(prev[k])* has the possibility of being updated to a higher value by selecting a different combination of activities.

So, *W(prev[k])* is optimal and we add the weight of a_k, i.e., w_k, to generate optimal *W(k)*.

So, *Optimal sum of first k scheduled activities* = w_k + *W(prev[k])*

Hence, to get the optimal value for weighted activity selection, we can simply use the maximum value out the above two cases, i.e.,

$$max\{W(prev[k]) + w_k, W(k-1)\}$$

Hence, the correctness of this recursion is proved.

(d) DYNAMIC PROGRAMMING APPROACH:

WEIGHTED_ACTIVITY_SELECTION_ALGORITHM

Input: Array of start times (*s*), Array of finish times(*f*), Array of weights(*w*)

PRE-PROCESSING STEP:

```
1. Sort the activities such that their finish times are in increasing
   order
2. Create an array prev, such that for each index 'i', it contains
   index of activity 'j' that is compatible with this activity.
   [i.e., activity with highest f_j such that f_j < s_i]
3. nums <- total number of activities
4. Create an empty array W[] of size nums
5. W[0] <- 0
6. find_max_weight(k):
7.       if k==0:
8.         then return W[0]
9.       if W[k] == "NULL"
10.      then
         W[k]=max{find_max_weight(prev[k])+w_k,find_max_weight(k-1)}
11.      RETURN W[k]
```

FINDING THE OPTIMAL SELECTION OF ACTIVITIES:

```
1. activity_list <- NULL
2. W[] <- Array containing pre-processed information for nums elements
3. optimal_selection(i):
4.       if i==0:
5.         then return
6.       if (W[prev(i)] + w_i)> W[i-1]:
7.         then activity_list.add(i)
8.              optimal_solution(i)
9.       Else
10.             Optimal_solution(i-1)
   print(activity_list)
   (to display the list of activities selected)
```

Analysis of the proposed algorithm:

PRE-PROCESSING

- Sorting can be done in *O(n lgn*) time complexity
- The array prev[] can be populated in *O(n lgn)* time complexity (using Binary Search)
- *find_max_weight(k)* function just fetches items from *prev[ ]* array or computes the value if it is *null*. When the value is *null*, two recursive calls are made to populate this entry in *prev[ ].*
- Even if in worst case, when the complete array is empty, there are at-most '*2*n' (n = nums)* calls made which make the time complexity of this function as O(n)
- Thus, overall time complexity of pre-processing step is ***O(n lgn)***

FINDING THE OPTIMAL SELECTION OF ACTIVITIES

- There are two recursive calls, only one of which is invoked in during a call to the function, so the total number of recursive calls will be at-most *n.*
- Therefore, the time complexity of this part of algorithm is ***O(n).***

WORKING EXAMPLE

i	***1***	***2***	***3***	***4***	***5***
s_i	1	2	3	4	5
f_i	2	4	10	8	12
w_i	10	20	10	5	100

Sorting the activities based on finish times (f_i)

s_i	1	2	4	3	5
f_i	2	4	8	10	12
w_i	10	20	5	10	100

Computing the *prev[ ]* array,

i	***0***	***1***	***2***	***4***	***3***	***5***
prev[i]	0	0	1	1	2	2

Computing the *W[ ]* array,

i	***0***	***1***	***2***	***4***	***3***	***5***
W[i]	0	10	30	30	35	130

activity_list : {1, 2, 5}

So, the optimal selection of activities has weight equal to value of W[5], i.e., ***130.***

3. [9 points] *Proving (or refuting) the greedy choice property*

Suppose *n* programs are stored on a magnetic tape (remember those?). Let *s*(*i*) be the size of program *i* and *f*(*i*) the frequency of use for program *i*. The time it takes to access a program is proportional to the sizes of all programs on the tape up to and including it. The goal is to minimize the total access time for all programs by finding an ordering of the programs on the tape. More formally, let π be a permutation of 1,...,*n* that describes the ordering: so if $\pi(1) = i$ then program *i* is the first on the tape. The total cost is:

$$\sum_{i=1}^{n} f(\pi(i)) \sum_{k=1}^{i} s(\pi(k))$$

There are three possible greedy algorithms for minimizing this cost:

(a) select programs in order of increasing *s*(*i*) (nondecreasing if we allow for equal sizes – I use increasing to avoid confusion)

(b) select programs in order of decreasing *f*(*i*)

(c) select programs in order of decreasing *f*(*i*)/*s*(*i*)

For each algorithm, either find a counterexample to show that it is not optimal or prove that it always gives an optimum solution.

Sol.

(a) Consider the following example,

i	*1*	*2*	*3*	*4*	*5*
f(i)	10	16	4	1	3
s(i)	10	20	8	5	30

Sorting the above array in the increasing order of s_i. The array gets transformed in the following way:

i	*1*	*2*	*3*	*4*	*5*
f(i)	1	4	10	16	3
s(i)	5	8	10	20	30

The total cost of picking the elements in this order

= 1*5 + (5+8)*4 + (5+8+10)*10 + (5+8+10+20)*16 + (5+8+10+20+30)*3

= 5 + 52 + 230 + 688 + 219

= **1194**

Now, let's consider the initial arrangement,

The total cost of picking the elements in the initial order

= 10*10 + (10+20)*16 + (10+20+8)*4 + (10+20+8+5)*1 + (10+20+8+5+30)*3

= 100 + 480 + 152 + 43 + 219

= **994**

Clearly, from this example, we can observe that there exists a solution which is better than the one which we obtain by considering the records in increasing order of *s(i)*.

(b) Consider the following example,

i	*1*	*2*	*3*	*4*	*5*
f(i)	10	16	4	1	3
s(i)	10	20	8	5	30

Sorting the above array in the increasing order of s_i. The array gets transformed in the following way:

i	*1*	*2*	*3*	*4*	*5*
f(i)	16	10	4	3	1
s(i)	20	10	8	30	5

The total cost of picking the elements in this order

= 20*16 + (20+10)*10 + (20+10+8)*4 + (20+10+8+30)*3 + (20+10+8+30+5)*1

= 320 + 300 + 152 + 204 + 73

= **1049**

Now, let's consider the initial arrangement,

The total cost of picking the elements in the initial order,

= 10*10 + (10+20)*16 + (10+20+8)*4 + (10+20+8+5)*1 + (10+20+8+5+30)*3

= 100 + 480 + 152 + 43 + 219

= **994**

Clearly, from this example, we can observe that there exists a solution which is better than the one which we obtain by considering the records in decreasing order of *f(i)*.

(c) Let us consider two programs stored on tape, one at position *π(i)* and another at position *π(i+1)*

So, first let's find the change in total cost due to swapping these two programs.

Let $a = \pi(i)$ and $b = \pi(i+1)$,

Net change in total cost by swapping programs at *a* and *b* is,

Change in net cost = (change in cost due to placing program *a* in place of program *b*)

+ (change in cost due to placing program *b* in place of program *a*)

Calculation for "*Change in cost due to placing program a in place of program b":*

Since, program *a* is moved to place of program *b*,

We'll have to traverse additional records which will be equal to size of program *b*, while accessing program *a*, this can be shown mathematically as,

Additional size to be traversed = $\sum_{k=1}^{i+1} s(\pi(k)) - \sum_{k=1}^{i} s(\pi(k))$

$= s(\pi(i+1))$

$= s(b)$

But this should be multiplied by factor equal to the frequency of the program, as each time the program is accessed, additional size must be traversed.

So, additional cost for this case = $s(\pi(i+1)) * f(\pi(i))$

$$= s(b)*f(a) \qquad \text{..... Eqn. (1)}$$

Calculation for "*Change in cost due to placing program b in place of program a":*

Since, we are promoting the program *b* to an earlier place (position to program *a*), so this change will be a negative quantity.

This is because, lesser programs need to be traversed to reach program *b*, and hence this quantity will be ***s(a)*f(b).***

Here, *s(a)* is the size of program *a*, which now, is not required to be traversed for accessing program *b*. And this amount of cost is saved for each access to program *b*, hence *s(a)* is multiplied with factor of *f(b)*.

So, additional cost for this case = $- s(\pi(i)) * f(\pi(i+1))$

$$= - s(a)*f(b) \qquad \text{..... Eqn. (2)}$$

From the above equations *(1)* and *(2)*, we can now calculate the net change in cost due to this swapping,

$$\textbf{Change in net cost} = s(b)*f(a) - s(a)*f(b) \qquad \text{.... Eqn. (3)}$$

Now, as per the instructions in the question, the programs are arranged in decreasing order of *f(i)/s(i)*

So, for any program *x* that comes before program *y*, we can say that

$$\frac{f(x)}{s(x)} > \frac{f(y)}{s(y)}$$

So, performing cross-multiplication on the above inequality, we get,

$f(x)*s(y) > f(y)*s(x)$ *[Since, both the quantities are positive, there is no change in the sign of inequality]*

So, $f(x)*s(y) - f(y)*s(x) > 0$

This is similar to the expression in the *eqn. (3)*, which denotes the change in cost for swapping two programs.

Hence, the net change in cost for swapping any two programs when they are arranged in decreasing order of *f(i)/s(i)* is ***positive***.

This proves that such an arrangement will give optimal solution as any swap operation results in increase in cost.