(Sol) HW03 Dynamic Programming

1. [4 points] Let $p_1, p_2, \cdots p_n$ be n programs to be stored on a disk. Program p_i requires s_i MBs of storage, and the capacity of the disk is D, where $s_1 + s_2 + \cdots + s_n > D$. Give an efficient algorithm to determine which programs to store on the disk so that as much of the disk as possible is utilized.

   a. (2 points) Provide the pseudo code of your algorithm.

      Answer: This problem is similar to a knapsack problem, where the weight of an interval = the weight of a program. Thus, a solution to the knapsack problem will suffice:
      $S[k, w]$ = the maximum amount of disk space that can be used for a disk of size w, if we only use pick programs in the range $s_1, \cdots, s_k$

      Finding $S[n, D]$ would give us the optimal solution.

      $$\begin{aligned} S[i,j] &= 0 && \text{if } i = 0 \\ &= S[i-1, j] && \text{if } s_i > j \\ &= \max(s_i + S[i-1, (j-s_i)], S[i-1, j]) && \text{otherwise} \end{aligned}$$

      We can fill this table from right to left and from top to bottom:
      The following function calculates the value of the array Take[], where Take[i] is true if and only if program p_i is in the optimal solution.

```
DiskUsage()
  for i = 1 to n do
    S[0,i] = 0
  for i = 1 to n do
    for j = 1 to D do
      if s[i] > j then
        S[i, j] = S[i − 1, j]
        TakeH[i, j] = false
      else if (s[i] + S[i−1,(j−s[i]) > S[i−1,j])then
        S[i,j] = s[i] + S[i−1,(j−s[i])]
        TakeH[i, j] = true
      else
        S[i, j] = S[i−1, j]
        TakeH[i, j] = false
```

```
w = D;
for i = n to 1 do
  Take[i] = TakeH[i, w]
  if TakeH[i, w] then
    w = w - s[i]
return Take
```

b. (2 points) What is the running time of your algorithm, in terms of n? Explain your answer.
Answer: The table is n × D, so it takes $\Theta(n \times D)$ to fill the table. Once the table is ready, it takes another $O(n)$ operations to find out whether each p_i is in the optimal solution or not.

2. [6 points] Weighted Interval Scheduling Problems

a. [2 points] Consider the Weighted Interval Scheduling Problem with a different objective. The weights of intervals now represent “penalty” (perhaps a fine we pay) – if we accept a request, there is no fine for that, but if we do not accept a request i, then we must pay penalty of w[i]. Hence, our goal is to minimize the sum of penalties of the unaccepted requests. As an example, see Figure 6.1 in KT with three requests. If we accept request 2 alone, the penalty will be 2 (as we pay 1 for request 1 and 1 for request 3), which is optimal. If we accept requests 1 and 3, we pay 3 (for request 2). Note that we can’t accept all requests because they are not compatible.

Describe very concisely how you can solve this problem efficiently, based on what we learned so far. That is, how can your algorithm minimize the overall penalty by accepting a compatible set of requests? No proof is needed.

Answer:

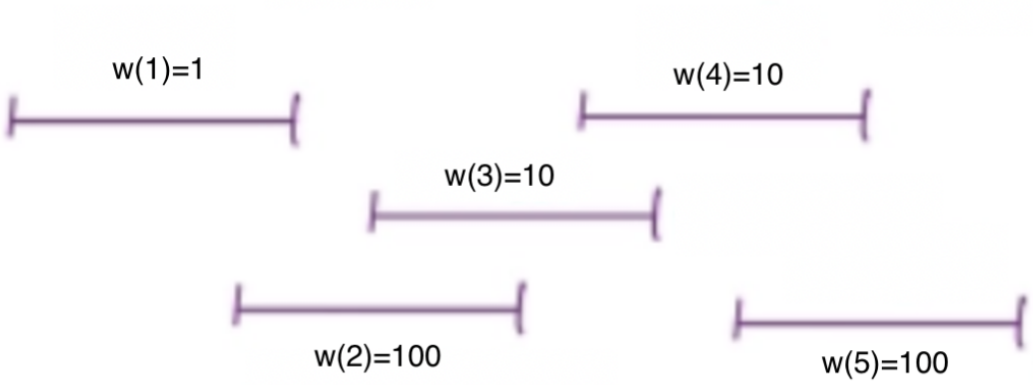
- In the WIS Problem, the objective is to ***maximize*** the sum of the weights of ***the selected intervals***.
- Notice that the sum of the weights of the selected & non-selected intervals is... always the same (it's simply the sum of all weights).
- To solve the new problem, we can first solve the WIS problem to obtain the maximum possible total weight (call it S); then we output (sum of all weights ***minus*** S).

b. [4 points] One motivation for solving the Interval Scheduling Problem or the Weighted Interval Scheduling Problem is when we have one shared resource (such as a meeting room or computing equipment). Now suppose that we have two identical, shared resources. This means that we can now accept two compatible sets of requests such that one set is handled by one resource and the other by the other resource (e.g., we
can assign a set of meetings to one meeting room and another set of meetings to the other).

Here is a reasonable (Greedy) algorithm using our DP algorithm as subroutine: First, find an optimal set of requests with maximum weight using the DP algorithm we learned (if there are multiple such sets, just pick an arbitrary one) – these requests are assigned to the first resource. Remove these requests from the original input, and run the DP algorithm again with the remaining requests – these are assigned to the second request.
Using Figure 6.1 in KT as an example: The first resource will accept request 2, and we'll solve the same problem again on two requests (request 1 and request 3) which will be assigned to the second resource. Here, we are able to accept all requests, and thus achieve total weight of 5.

Unfortunately, this Greedy-DP algorithm is incorrect. Find a counterexample where the proposed algorithm fails to find an optimal solution. Clearly describe an input instance (n requests with their start time, finish time, and weight – a drawing is fine, and is recommended, actually), how this algorithm would assign requests to two resources, and what the optimal solution to the input is.
(Hint:.5=n ro 4=n ekil selpmaxe llams yrT)

- Here is one example with n = 5:
    - The optimal solution is to assign intervals {1, 3, 5} to the one resource and {2, 4} to the other resource.
    - The proposed Greedy-DP algorithm will first accept {2, 5} and assign them to the first resource (that's the optimal solution so far). It will then have to choose either {1, 3} or {1, 4} in the second phase, and therefore this algorithm can only accept four intervals.


w(1)=1
w(4)=10
w(3)=10
w(2)=100
w(5)=100