Solutions for HW 1 Dynamic Programming Problems

[DPV] Problem 6.1 – Max contiguous subsequence

Solution: Let's first solve the problem of finding the maximum sum obtainable by a contiguous subsequence, and then we'll see it's easy to modify that algorithm to output a valid subsequence.

For solving the problems using dynamic programming, we first identify the subproblems and then we try to express a recurrence for the solution to the subproblem in terms of smaller subproblems. Our first attempt is often to consider *prefixes* for the subproblem. For the first i numbers, find the maximum contiguous subsequence within it. However, one will realize that the recurrence is hard to write down because we have to maintain the property that the solution is contiguous. As in the longest increasing subsequence problem, we have to strengthen the subproblem so that we consider the best solution that includes the number a_i.

We define the subproblem as:

$S(i) =$ the maximum sum attainable by a contiguous subsequence from the list of numbers $a_1, a_2, a_3, ..., a_i$ with the extra constraint that the contiguous subsequence has to end with a_i.

The reason to force the subproblem to have this extra condition is that, just as for the longest increasing subsequence problem that we saw in class, it is necessary in order to write a recursive formula for $S(i)$ in terms of $S(1), \dots, S(i-1)$.

Let's look at the example in the book, $A = [5, 15, -30, 10, -5, 40, 10]$. Then, $S(1) = 5$ since 5 itself is the only option. $S(2) = 20$ since $5, 15$ has larger sum than 15 by itself. Similarly, $S(3) = -10$ since $5 + 15 - 30 = -10$ is larger than -30 by itself. Note, $S(4) = 10$ since that is larger than $5 + 15 - 30 + 10 = 0$. The whole table is $S = [5, 20, -10, 10, 5, 45, 55]$.

Notice that to compute $S(i)$ we either consider adding a_i to the optimal subsequence ending at a_{i-1} or we consider a_i by itself. Hence, the recurrence is the following:

$$S(i) = \max\{a_i, a_i + S(i-1)\}.$$

It is easy to see that once we define these subproblems, the final solution to the original problem is the best solution among $S(1), S(2), ..., S(n)$ which can be computed in $O(n)$ time assuming that all the subproblems are solved. The pseudocode for filling the table is below.

To obtain a valid subsequence, as in the longest increasing subsequence problem that we did in class, we just have to "backtrack" in the table S to determine if the optimal subsequence only includes a_i, or if it continues to the optimal subsequence ending at a_{i-1}. To this end we add use the table $prev()$ to keep track of which value achieves the maximum in the recurrence.

Algorithm 1 Max sum of contiguous subsequence$(a_1, a_2, \ldots, a_n)$

```
S(0) = 0.
for i = 1 to n do
    if S(i − 1) + a_i ≥ a_i then
        S(i) = S(i − 1) + a_i
        prev(i) = i − 1
    else
        S(i) = a_i
        prev(i) = NULL
    end if
end for
max = 1
for i = 2 to n do
    if S(i) > S(max) then
        max = i
    end if
end for
return S(max)
```

The running time is $O(n)$.

To output a contiguous subsequence obtaining the maximum sum we add the following to the algorithm.

Algorithm 2 Outputting a contiguous subsequence of maximum sum

```
i = max
output(i)
while prev(i) ≠ NULL do
    i = prev(i)
    output(i)
end while
```

[DPV] Problem 6.2 – Hotel stops
Solution:

As in Problem 1, we strengthen the subproblem so that we consider the best subsequence which includes hotel i.

Therefore, we define the following subproblem.

For $1 \leq i \leq n$, let

$$P(i) = \begin{array}{c}\text{minimum penalty obtainable for the trip} \\ \text{from mile 0 to mile } a_i \text{ with the last stop at hotel } i.\end{array}$$

To figure out the recurrence, we consider the choices for the last hotel stop prior to i. Say that this penultimate stop is at hotel k which is at mile a_k. Then the cost for the last segment of the trip is $(200-(a_i-a_k))^2$, and it's $P(k)$ for the beginning segment of the trip ending at hotel k. Therefore, the total cost for this trip with a penultimate stop at the k-th hotel is $P(k) + (200 - (a_i - a_k))^2$. We then try all possibilities for k, so all k where $0 \leq k \leq i - 1$. Note, $k = 0$ corresponds to having no stops prior to the i-th hotel. Finally, the recurrence is

$$P(i) = \min_k \{P(k) + (200 - (a_i - a_k))^2 : 0 \leq k \leq i - 1\}$$

This says that $P(i)$ is the minimum over k, where k is restricted to be at least 0 and at most $i-1$, and we're minimizing the quantity $P(k)+(200-(a_i-a_k))^2$. The base case is $P(0) = 0$.

Below is the pseudocode for filling the table. To implement the minimum over k, we first set $P(i)$ to the value from the $k = 0$ case, then we try other choices of k and redefine $P(i)$ if a smaller cost is found.

To find a set of locations with the minimum penalty we add the $prev(i)$ to store the k which achieves the minimum in the recurrence, and then we backtrack.

The running time is $O(n^2)$.

Algorithm 3 Hotel stops($a_1, a_2, \ldots, a_n$)

```
P(0) = 0
for i = 1 to n do
    P(i) = (200 − a_i)^2
    prev(i) = NULL
    for k = 1 to i − 1 do
        if P(i) > P(k) + (200 − (a_i − a_k))^2 then
            P(i) = P(k) + (200 − (a_i − a_k))^2
            prev(i) = k
        end if
    end for
end for
{**P(n) is the minimum penalty obtainable**}
{**The following outputs a set of locations obtaining the minimum penalty**}
i = n
output(i)
while prev(i) ≠ NULL do
    i = prev(i)
    output(i)
end while
```

[DPV] Problem 6.3 – YuckDonald's

Solution: Again this problem is exactly the same flavor as Problems 1 and 2. We define the subproblem as:

$$L(i) = \begin{array}{c} \text{maximum profit from a valid subset of locations } m_1, m_2, ..., m_i \\ \text{with the extra constraint that } m_i \text{ has to be included.} \end{array}$$

The final solution to the problem is therefore $\max_i L(i)$.

The base case is $L(0) = 0$. To define the recurrence for $L(i)$, since we are putting a restaurant at location m_i, we gain profit p_i from it, and then the penultimate location must be at least k miles away. Hence, the penultimate location can only be those j where $m_j \leq m_i - k$. We try all possibilities for this penultimate location m_j, and the maximum profit we obtain for a subset of locations $1, \ldots, j$ is captured in $L(j)$. Hence, if the penultimate location is m_j the total profit we obtain is $p_i + L(j)$. Therefore, the recurrence is the following:

$$L(i) = p_i + \max_j \{L(j) : m_j \leq m_i - k\}.$$

Here is the pseudocode for filling the table:

Algorithm 4 YuckDonalds$(a_1, a_2, \ldots, a_n)$

```
L(0) = 0.
for i = 1 to n do
    L(i) = p_i.
    for j = 1 to i − 1 do
        if m_j ≤ m_i − k then
            if L(i) < L(j) + p_i then
                L(i) = L(j) + p_i
            end if
        end if
    end for
end for
return max_i L(i)
```

The running time is $O(n^2)$.

Alternative Solution:

One can also solve the problem by defining the subproblem as follows:

$P(i)$ = maximum profit from a valid subset of locations $m_1, m_2, ..., m_i$

Hence, we have dropped the extra constraint that m_i has to be included. In this case, the final solution to the problem is therefore simply $P(n)$.

To figure out the recurrence, we have two cases, either m_i is included or it is not included. If it is not included, then the best subset of locations $m_1, \ldots, m_i$ is the same as the best subset of locations $m_1, \ldots, m_{i-1}$. Hence, in this case, $P(i) = P(i-1)$. If location m_i is included then the penultimate location must be at least k miles away. Let $last(i)$ denote the last possible location that is at least k miles from m_i, in other words:

$$last(i) = \max\{\ell : m_\ell \leq m_i - k\}.$$

Now if location m_i is included, then the remaining solution must be a subset of locations $m_1, \ldots, m_\ell$ where $\ell = last(i)$. Hence, in this case where location

m_i is included, we have that $P(i) = p_i + P(last(i))$. Therefore, the recurrence is the following:

$$P(i) = \max\{P(i-1), p_i + P(last(i))\}$$

It is easy to modify the earlier pseudocode to obtain an $O(n^2)$ time solution. In fact, by first calculating $last(i)$ for $i = 1 \to n$, and then calculating $P(i)$ for $i = 1 \to n$, one obtains an $O(n)$ time solution.

Here is the pseudocode for the faster solution:

Algorithm 5 FasterYuckDonalds($a_1, a_2, \ldots, a_n$)

```
a_0 = 0.
ℓ = 0.
for i = 1 to n do
    while ℓ < i − 1 and m_{ℓ+1} ≤ m_i − k do
        ℓ = ℓ + 1.
    end while
    last(i) = ℓ.
end for
P(0) = 0.
for i = 1 to n do
    P(i) = max{P(i − 1), p_i + P(last(i))}.
end for
return P(n)
```

[DPV] Problem 6.17 – Making change I
Solution:

In this problem, you have n denominations $x_1, \dots, x_n$ (with unlimited supply of each) and a value V, and you are asked to determine in $O(nV)$ time whether there is a set of coins with total value V. This problem is similar to the knapsack problem (with repetition).

We make a one dimensional table. For $0 \leq w \leq V$, let

$$S(w) = \{\texttt{TRUE} \text{ or } \texttt{FALSE} \text{ whether there is a subset of coins with total value } w\}.$$

By considering the last coin used, we get that $S(w)$ is `TRUE` if there is a denomination $1 \leq i \leq n$ where $S(w - x_i)$ is `TRUE`. Hence, we get the following recurrence, where $\bigvee$ denotes OR. For $0 \leq w \leq V$,

$$S(w) = \bigvee_i \{S(w - x_i) : 1 \leq i \leq n, x_i \leq w\}$$

This yields an $O(nV)$ time algorithm with 2 for-loops.

Algorithm 6 Coin Changing I

$S(0) = \texttt{TRUE}$.
for $j = 1$ to v **do**
    $S(j) = \texttt{FALSE}$.
end for
for $i = 1$ to v **do**
    for $j = 1$ to n **do**
        if $i - x_j \geq 0$ **then**
            $S(i) \leftarrow S(i - x_j) \vee S(i)$
        end if
    end for
end for
return $S(v)$

[DPV] Problem 6.18 – Making change II

Solution: This problem is very similar to the knapsack problem without repetition that we saw in class.

First of all, let's identify the subproblems. Since each denomination is used at most once, consider the situation for x_n. There are two cases, either

- We do not use x_n then we need to use a subset of $x_1, \ldots, x_{n-1}$ to form value v;
- We use x_n then we need to use a subset of $x_1, \ldots, x_{n-1}$ to form value $v - x_n$. Note this case is only possible if $x_n \leq v$.

If either of the two cases is `TRUE`, then the answer for the original problem is `TRUE`, otherwise it is `FALSE`. These two subproblems can depend further on some subproblems defined in the same way recursively, namely, a subproblem considers a prefix of the denominations and some value.

We define a $n \times v$ sized table D defined as:

$$D(i, j) = \{\texttt{TRUE} \text{ or } \texttt{FALSE} \text{ where there is a subset of the coins of denominations } x_1,...,x_i \text{ to form the value } j.\}$$

Our final answer is stored in the entry $D(n, v)$.

Analogous to the above scenario with denomiation x_n we have the following recurrence relation for $D(i, j)$. For $i > 0$ and $j > 0$ then we have:

$$D(i, j) = \begin{cases} D(i-1, j) \vee D(i-1, j-x_i) & \text{if } x_i \leq j \\ D(i-1, j) & \text{if } x_i > j. \end{cases}$$

(Recall, $\vee$ denotes Boolean OR.)

The base cases are $D(0, 0) =$ `TRUE` and for all $j = 1, 2, \ldots, v$, $D(0, j) =$ `FALSE`.

The algorithm for filling in the table is the following.

Each entry takes $O(1)$ time to compute, and there are $O(nv)$ entries. Hence, the total running time is $O(nv)$.

Algorithm 7 Coin Changing II

```
D(0,0) = TRUE.
for j = 1 to v do
    (0,j) = FALSE.
end for
for i = 1 to n do
    for j = 0 to v do
        if x_i ≤ j then
            D(i,j) ← D(i−1,j) ∨ D(i−1,j−v_i)
        else
            D(i,j) ← D(i−1,j)
        end if
    end for
end for
return D(n,v)
```