Exercise VI, Algorithms 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

For more exercises on dynamic programming and video explanations of solutions, see *http://people.csail.mit.edu/bdean/6.046/dp/*

Solve New Problems Using Dynamic Programming

1 *(25 pts)* **Restaurant placement.** Justin Bieber has surprisingly decided to open a series of restaurants along the highway between Geneva and Bern. The n possible locations are along a straight line, and the distances of these locations from the start of the highway in Geneva are, in kilometers and in arbitrary order, $m_1, m_2, \ldots, m_n$. The constraints are as follows:

- At each location, Justin may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.
- Any two restaurants should be at least k kilometers apart, where k is a positive integer.

As Justin is not famous for his algorithmic skills, he needs your help to find an optimal solution, i.e., **design and analyze** an *efficient* algorithm to compute the maximum expected total profit subject to the given constraints.

Solution: Let M be the array containing the distances of the restaurants from Geneva, in increasing order; we can compute this array in $O(n \log n)$ time (e.g. using mergesort, heapsort, etc.), and let p be the array whose i-th element contains the profit we gain by opening a restaurant at position $M[i]$. Now consider the opening of the leftmost restaurant: since it is the leftmost restaurant, there will be no restaurant closer to Geneva. Furthermore, opening this restaurant restricts the set of locations where we can open other restaurants, since restaurants must be at least k kilometers apart. Thus, given that the leftmost restaurant is opened at a particular location i, we can use the following (recursive) strategy to open the most profitable set of restaurants. Let $c[i]$ be the entry in array c containing the optimal profit for the possible set of restaurant locations whose leftmost restaurant is opened at location $M[i]$. Formally we have:

$$c[i] = p[i] + \max_{i<j\leq n : M[j] \geq M[i]+k} c[j].$$

We implement this recursive formulation efficiently using the Bottom-up approach:

```
RESTAURANTS($M, p$)
1. $c[n] \leftarrow p[n]$
2. for $i = n$ to 1
3.       $l \leftarrow 0$
4.       for $j = i + 1$ to $n$
5.             if $c[j] > l$ and $M[j] \geq M[i] + k$
6.                   $l \leftarrow c[j]$
7.             end if
8.       end for
9.       $c[i] \leftarrow p[i] + l$
10. end for
11. return $\max_{1 \leq i \leq n} c[i]$
```

Let us analyze the running time of the above algorithm: the two nested loops will cause lines 5-7 to be executed $\Theta(n^2)$ times. Line 11 requires $\Theta(n)$ operations, while sorting requires $O(n \log n)$ operations. Hence, the total running time is $\Theta(n^2)$.

Note that we can design an algorithm with $O(n \log n)$ running time too. As a preprocessing step, we sort the possible locations according to their distances, then, we calculate the nearest possible location on the right of m_i which is at least k kilometers away. We denote such location for each i by $b[i]$. If no such location exists, we set $b[i] = n + 1$. The algorithm below returns the array b in $\Theta(n)$ time.

```
LeftPossible($M$)
1. pointer = 1
2. for $i = 1$ to $i = n$
3.       while pointer <= n+1 and $M[pointer] - M[i] < k$
4.             $pointer \leftarrow pointer + 1$
5.       $b[i] \leftarrow pointer$
6. return $b$
```

Let $d[i]$ be the maximum profit that can be achieved from opening restaurants from m_i to m_n. At each position, we can either open a restaurant or keep it closed. If we open a restaurant, our profit would be $p[i] + d[b[i]]$. If we do not open a restaurant in m_i, then our profit would be $d[i + 1]$. Here we implement this algorithm:

```
FastRestaurant($M, p$)
1. $d[n + 1] \leftarrow 0$
2. b = LeftPossible($M$)
3. for $i = n$ to $i = 1$
4.       $d[i] = \max\ (d[i + 1], d[b[i]] + p[i])$
5. return $b[1]$
```

The total running time for this algorithm is $O(n \log n) + \Theta(n)$.

2 (half *, Problem 15-1) **Longest simple path in a directed graph.**

Suppose that we are given a directed acyclic graph $G = (V, E)$ with real valued edge weights and two distinguished vertices s and t. Describe a dynamic programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?

Solution: First of all, note that all paths in DAG(directed acyclic graph) are simple. If there are two parts of the path then they cannot visit the same vertex, except when this vertex is the end and the beginning of the first and the second subpaths respectively.

The following observation gives an idea to the algorithm. Let p be the longest path from s to t and let v be the next vertex on this path after s. Then the longest path p' from v to t is a part of p. If not, let p'' be longer than p', then the path from s, which goes to v and then continues with p'', is the path from s to t, which is longer than p.

Let $dist[s]$ denote the weight of the longest path from s to t.

$$dist[s] = \begin{cases} 0 & \text{if } s = t, \\ max_{(s,v)\in E}(w(s,v) + dist[v]) & \text{otherwise.} \end{cases}$$

To solve the problem the following algorithm can be used. *next* is an array, which stores the next vertex on the longest path. Also, if there is a value, then the subproblem for this vertex has been already solved. Before it we have to initialise both arrays: *dist* with 0 and *next* with *null* values.

```
LONGEST-PATH(G,S,T, DIST, NEXT)
1   if $s == t$
2     $dist[s] = 0$
3     return $(dist, next)$
4   elseif $next[s] \neq null$
5     return $(dist, next)$
6   else
7     for each vertex $v \in G.Adj[s]$ (adjacent vertex)
8       $(dist, next)$ = LONGEST-PATH(G,V,T,DIST,NEXT)
9       if $w(s,v) + dist[v] \geq dist[s]$
10          $dist[s] = w(s,v) + dist[v]$
11          $next[s] = v$
12    return $(dist, next)$
```

At the end $dist[s]$ stores the weight of the longest path from s to t. To print the path we need only to go by the links of the array *next* starting from s until we reach t.

The running time of this algorithm is $O(V)$ to initialize both arrays, $O(E)$ for the $Longest-Path$ part (we call it once for each edge of the vertex, thus we call it no more than E times) and $O(V)$ to restore the path. Thus, in sum it requires $O(V + E)$.

3 (*) **Knapsack Problem** Suppose that you are going on a beautiful hike in the Swiss Alps. As always, you are faced with the following problem: your knapsack is too small to fit all the items that you wish to bring with you. As items are of different importances and have different sizes, you would like to maximize the total value of the items that you can bring with you. Formally, we can define the "packing" problem as follows:

INPUT: A knapsack of capacity C and n items where item $i = 1, 2, \ldots, n$ has value $v_i \geq 0$ and size $s_i > 0$.

OUTPUT: A subset of items S that maximizes $\sum_{i\in S} v_i$ (the total value) subject to $\sum_{i\in S} s_i \leq C$ (the total size of the packed items is at most the capacity).

3a (The Fractional Knapsack Problem) Suppose that your items are divisible, i.e., you can pack a fraction $f \in [0, 1]$ of an item i and in that case it will give you a profit of $f \cdot v_i$ and occupy a space of $f \cdot s_i$ in the knapsack. Give a greedy algorithm for this case that runs in time $O(n \log n)$.

Solution: In Fractional Knapsack Problem we can pack a part of an item. This gives a simple greedy solution. To maximize the total value of the knapsack we simply need to take items, which value of a unit of weight are the biggest ones. Let rank each item i by the value of the unit of weight v_i/s_i and fill in the knapsack with items which have greater rank until there are no items left or a knapsack is full.

```
FRACTIONAL-KNAPSACK(v,c,C)
1  Sort items by $v_i/s_i$. Assume that $v_i/s_i \geq v_{i+1}/s_{i+1}$ for all $i$
2  $load = 0$
3  $i = 1$
4  while $i \leq n$ and $load \leq C$
5     if $C - load \geq s_i$
6        take the whole item $i$ and $load = load + s_i$
7        else take $C - load/s_i$ of item $i$ and $load = C$
8     $i = i + 1$
```

Time of the algorithm is $O(n\ lg\ n)$ for sorting and $O(n)$ for the second part of the algorithm.

3b Show that the greedy algorithm fails when the items are indivisible. How bad can the profit of the solution returned by the greedy algorithm be compared to an optimal solution?

Solution: In Knapsack Problem (non-fractional) greedy algorithm can leave a lot of empty space in the knapsack. Assume there are 2 items: first one has a size 1 and a cost v, and the second one with size C and a cost $C * (v - 1)$. The greedy algorithm will choose the first item and there will not be enough space for the second one. The optimal solution is to take the second item (here it is assumed that v and C are big).

In the worst case the greedy algorithm leaves practically the whole knapsack empty and it can be up to C times worse than the optimal algorithm.

3c Design a dynamic programming algorithm to solve the Knapsack Problem. Your algorithm should run in time $O(nC)$.

Solution: We can recursively define the value of an optimal solution by the following formula:

$$c(i,s) = \begin{cases} 0 & \text{if } i = 0 \text{ or } s = 0, \\ c(i-1,s) & \text{if } s_i > s, \\ max(v_i + c(i-1, s-s_i), c(i-1,s)) & \text{otherwise,} \end{cases}$$

where $c(i,s)$ is an optimal solution for the set of items $\{1...i\}$ and the size of a knapsack s. The last case says that the optimal solution can either take item i and items from the optimal solution $c(i-1, s-s_i)$, or don't take item i, and in this case it is the same as the optimal solution for $c(i-1,s)$.

The algorithm fills the table $c[0..n, 0..C]$, each cell stores the best value. The table is filled row by row. At the end the cell $c[n,C]$ has the optimal value for the problem. To find the set of items we need to take, we trace this table back from $c[n,C]$. If $c[i,s] = c[i-1,s]$ then we do not take item i and continue tracing from $c[i-1,s]$. Otherwise we take item i and continue from $c[i-1, s-s_i]$. Here is the algorithm:

```
DYNAMIC-KNAPSACK(n,v,s,C)
for $j = 0$ to $C$
    $c[0, j] = 0$
for $i = 1$ to $n$
    $c[i, 0] = 0$
for $i = 1$ to $n$
    for $j = 1$ to $C$
        if $s_i \leq j$
            $c[i, j] = max(v_i + c[i-1, j-s_i], c[i-1, j])$
            else $c[i, j] = c[i-1, j]$
$i = n$
$j = C$
while $i > 0$
    if $c[i-1, j] \neq c[i, j]$
        print $i$
        $j = j - s_i$
    $i = i - 1$
```

The algorithm takes $\Theta(nC)$ time to fill the table and $\Theta(n)$ time to trace the solution.