CS 577-2: Introduction to Algorithms Fall 2020

Solutions to Midterm Exam 1

Instructor: Dieter van Melkebeek

Part (a)

We can test whether a given candidate capacity $c \in \mathbb{N}$ is sufficient by simulating the optimal schedule day by day and making sure we never have a water shortage (because the demand w_i exceeds the capacity c) and do not need more than f fills in total. This can be done in time $O(n)$. For completeness we provide the pseudocode below, where the variable t keeps track of the number of fills thus far, and W the current water level of the tanks.

Algorithm 1 Capacity-Suffices($n, r, c, w_1, \ldots, w_n$)

Input: $n, f, c \in \mathbb{N}$, $w_i \in \mathbb{R}$ for $i \in [n]$

Output: whether capacity c is sufficient to accommodate the demands $w_1, \ldots, w_n$ for n successive days with complete fills at the start of at most f days including the first day.

```
1: $W \leftarrow c$
2: $t \leftarrow 1$
3: for $i \leftarrow 1$ to $n$ do
4:     if $w_i > c$ or ($w_i > W$ and $t \geq f$) then
5:         return false
6:     if $w_i \leq W$ then
7:         $W \leftarrow W - w_i$
8:     else
9:         $W \leftarrow c - w_i$
10:        $t \leftarrow t + 1$
11: return true
```

Note that if capacity c suffices then every larger capacity also suffices. Thus, we can determine the smallest capacity that suffices using a binary search.

As the daily demand is no larger than 100 and there are n days, a capacity of $100n$ definitely suffices. Therefore, the range for c can be limited to all nonnegative integers up to $100n$. The resulting binary search takes $\lceil \log_2(100n + 1) \rceil = O(\log n)$ rounds. Each round involves running Capacity-Suffices once and takes $O(n)$ time. Thus, the overall running time is $O(n \log n)$.

Part (b)

We can apply the principle of optimality based on the fact that a refill on day d breaks up the problem into two independent subproblems of the same type, namely the first $d - 1$ days and the last $n - d$ days.

Moreover, if we consider the *first* refill d (if any), then the loss in revenue during the first $d - 1$ days is ℓ_1. It only remains to add the minimum total loss in revenue for the subproblem defined by the last $n - d$ days. Note that having the first refill on day d is feasible iff the total demand during the first $d - 1$ days does not exceed the capacity c.

The case of no refill can be considered as having a refill on day $n+1$ at no cost.

The result is the minimum over all feasible choices $d \in \{2, \ldots, n+1\}$ for the first refill.

The subproblems that arise throughout the recursion all correspond to suffixes of $[n]$. This leads to the following subproblems for $k \in [n+1]$: $\mathrm{OPT}(k)$ denotes the minimum total loss in revenue for the period consisting of days $\{k, \ldots, n\}$, or ∞ if there is no feasible schedule. Note that the latter will happen iff at least one of the daily demands exceeds the capacity c.

By the above discussion, we have the following recurrence:

$$\mathrm{OPT}(k) = \min \left(\infty, \min_{k<d\leq n+1 : \sum_{i=k}^{d-1} w_i \leq c} (\ell_k + \mathrm{OPT}(d)) \right). \tag{1}$$

The recurrence allows us to compute the values $\mathrm{OPT}(k)$ from $k = n$ down to $k = 1$ using the initialization $\mathrm{OPT}(n+1) = 0$. We return $\mathrm{OPT}(1)$.

Each application of the recurrence involves taking the minimum over $O(n)$ terms. Checking the condition $\sum_{i=k}^{d-1} w_i \leq c$ for a given term takes time $O(n)$ by itself. However, by keeping track of a running sum, the total time for checking the condition can be kept to $O(n)$ for a single application of the recurrence. Thus, the running time per application of the recurrence is $O(n)$. As there are $O(n)$ applications, the overall running time is $O(n^2)$.

For completeness, here is pseudocode for evaluating (1).

Algorithm 2

```
1: $L \leftarrow \infty$
2: $W \leftarrow c$
3: $d \leftarrow k+1$
4: while $d \leq n+1$ cand $w_{d-1} \leq W$ do
5:     $L \leftarrow \min(L, \mathrm{OPT}(d))$
6:     $W \leftarrow W - w_d$
7:     $d \leftarrow d+1$
8: return $\ell_k + L$
```

Apart from the memory space to store the n values of the array OPT, the evaluation of (1) as described above only takes space $O(1)$. Thus, the overall space need is $O(n)$.

Alternate solution The problem can be cast as a shortest path problem in the graph $G = (V, E)$, where $V = [n+1]$ and there is an edge (k, d) with weight ℓ_k for each $1 \leq k < d \leq n+1$ such that $\sum_{i=k}^{d-1} w_i \leq c$. The answer to part (b) is the minimum length of a path from 1 to $n+1$ (or ∞ if there is no such path).

As the graph G is a DAG, the shortest path problem can be solved in time $O(|V|+|E|) = O(n^2)$ multiplied with the time needed to compute the weight of an edge, which is $O(n)$, resulting in $O(n^3)$ time overall. Similar to the other solution, the amount of time spent in computing the weight of the edges can be reduced to $O(n^2)$ total, resulting in an overall running time of $O(n^2)$. The amount of space needed is $O(|V|) = O(n)$ plus the space needed to compute the weight of an edge, which is $O(1)$, for a total of $O(n+m)$ space.

Part (c)

We use the OPT notation from part (b). We only need to consider the case where $\mathrm{OPT}(1) < \infty$; there is no optimal schedule otherwise.

The first fill day is always day 1. The first refill day d in an optimal schedule can be constructed by figuring out a choice of d on the right-hand side of (1) that yields the minimum for $k = 1$ and satisfies the constraint that $\sum_{i=k}^{d-1} w_i \leq c$; there is no refill if $d = n + 1$ yields the minimum. If we search for the first $d > 1$ for which $\mathrm{OPT}(1) = \ell_1 + \mathrm{OPT}(d)$, the constraint is guaranteed to be satisfied. If $d \leq n$, we do a refill on day d, and proceed with $k = d$ to find the second refill day in the same way, etc., until we hit $d = n + 1$.

For completeness, here is pseudocode implementing this approach.

Algorithm 3

```
1: $S \leftarrow \emptyset$
2: $k \leftarrow 1$
3: $d \leftarrow 2$
4: while $d \leq n + 1$ do
5:     if $\mathrm{OPT}(k) = \ell_k + \mathrm{OPT}(d)$ then
6:         $S \leftarrow S \cup \{k\}$
7:         $k \leftarrow d$
8:     $d \leftarrow d + 1$
9: return $S$
```

Each use of an OPT value can be replaced by a call to the corresponding instance of problem (b). By storing the OPT values, the number of calls to the blackbox for (b) is no more than the different OPT values, which is $n+1$. As d increases by one in each iteration of the pseudocode and the amount of work per iteration outside of the calls is $O(1)$, the total time spent outside of the calls is $O(n)$.

Alternate solution The process can be viewed as finding a path from 1 to $n+1$ in the subgraph $G' = (V, E')$ of the DAG G described in the alternate solution to (b) consisting only of those edges (k, d) for which $\mathrm{OPT}(k) = \ell_k + \mathrm{OPT}(d)$. The DAG G' has the special property that every path starting from 1 can be completed to end up in $n + 1$. It follows that such a path can be found in time $O(n)$ with blackbox access to OPT.