CS 577: Introduction to Algorithms 10/26/2015

Homework 6 Solutions (Review Problems)

Instructor: Dieter van Melkebeek TA: Bryce Sandlund

Problem 1

We first discuss the maximization problems and then the feasibility problems.

Maximization problems In the subsequence case, the maximum sum is exactly equal to the sum of all the positive elements in A, which can be computed trivially in linear time with a single sweep through A. If we instead wish to find the maximum subarray sum, we turn to a dynamic programming approach.

In order to solve this problem, we find for every position in the array the maximum sum of a subarray that ends at that position, and then take the maximum of all those values. Consider position i. There are two possibilities for a best subarray (that is, one of maximal sum) that ends at position i: either it is empty (in which case the maximum sum is 0), or it is $A[k..i]$ for some $k \leq i$. If the latter, then $A[k..i-1]$ must be a best subarray that ends at $i-1$ since otherwise, replacing $A[k..i-1]$ with a best subarray would increase the total sum of elements in $A[k..i]$. The maximum of the two possibilities gives us the best value for subarrays ending at position i.

This suggests that given the best value for subarrays that end at $i-1$, we can compute the best value for a subarray that ends at i with a constant number of operations. The following specification and recurrence for OPT' formalizes this strategy, where OPT' is a one-dimensional table: For $1 \leq i \leq n$:

$$\begin{aligned}\text{OPT}'(i) &\doteq \text{the maximum sum of a subarray of } A \text{ that ends at index } i\\ &= \max(0, A[i] + \text{OPT}'(i-1))\end{aligned}$$

For the base case we simply set $\text{OPT}'(0) = 0$, and we can fill out the table in increasing order of i. The final output is then $\max_{1\leq i\leq n} \text{OPT}'(i)$, and can be computed in parallel with OPT' as follows:

$$\begin{aligned}\text{OPT}(i) &\doteq \text{the maximum sum of a subarray of } A[1..i]\\ &= \max(\text{OPT}(i-1), \text{OPT}'(i)),\end{aligned}$$

where we set $\text{OPT}(0) = 0$. The final answer is $\text{OPT}(n)$.

Since each of the n entries of OPT' and OPT only requires a constant number of operations, the entire procedure takes linear time, as desired. In terms of memory, as we only need $\text{OPT}'(i-1)$ and $\text{OPT}(i)$ in order to compute $\text{OPT}'(i)$ and $\text{OPT}(i)$, we only need a constant amount of memory space.

Feasibility problems In some sense, determining whether a particular v is feasible is harder than finding the best possible value for v. The best value for the first i elements of A is a simple function of only the best value for the first $i-1$, but in the feasibility case, there is no notion of "best" since the only thing that matters is whether or not we can get exactly v in the end.

For the subsequence version of the problem, we will draw on knapsack for inspiration–in particular, by letting one of the dimensions of the OPT table be indexed by values. Given a particular value w, if there exists a subsequence of $A[1..i]$ that sums to w then one of two possibilities must be true: either there is a subsequence of $A[1..i-1]$ that sums to w (if $A[i]$ is not part of the sum), or there is one that sums to $w - A[i]$ (if $A[i]$ is part of the sum). This means that we can reduce the feasibility of a particular w over $A[1..i]$ to the feasibility of all possible values over $A[1..i-1]$, giving rise to the following recurrence for $1 \le i \le n$:

$$\begin{aligned}\mathrm{OPT}(i, w) &\doteq \text{true if there exists a subsequence of } A[1..i] \text{ that sums to } w \text{ or false o/w} \\ &= \mathrm{OPT}(i-1, w) \vee \mathrm{OPT}(i-1, w - A[i]),\end{aligned}$$

where we consider $\mathrm{OPT}(0, w)$ for every possible w as the base cases; these are all false except for $\mathrm{OPT}(0,0)$. The OPT table is of dimension $(n+1) \times L$, where $L = 1 + 2\sum_{i\in[n]} |A[i]|$ is such that the possible values for w range from the most negative sum of elements in A to the most positive. The table can be filled out in increasing order of i, and as per the specification for OPT, the final answer can be found in $\mathrm{OPT}(n, v)$. Since each entry of OPT takes a constant number of operations to compute, the entire table can be filled in time $O(nL)$, which is pseudopolynomial in the length of the input (and hence much worse than the linear-time algorithm for the corresponding maximization problem). As we can fill out the table row-by-row, we only need to keep track of the previous array and the current array.

The subarray version of the problem is easier. The brute force algorithm takes $O(n^3)$ time–there are $O(n^2)$ subarrays of A, and checking whether each sums to v takes $O(n)$ time for a total of $O(n^3)$. However, computing this sum for every subarray is wasteful, since we will find that we are frequently summing the same chains of elements over and over again. By processing the subarrays in increasing order of length and saving the results, we can compute the sum of each subarray with only a single addition, giving us a total time of $O(n^2)$. Formalizing this algorithm in the DP framework is left as an exercise for the reader.

Problem 2

We can reduce an instance of this problem to an easier instance of the same problem by considering the first decision we need to make: Do we get lollies on the first day or not? If we do, then we get ℓ_1 lollies the first day, and we additionally need to find the maximum number of lollies we can get during days $1 + k_1 + 1$ through n. Otherwise, we do not get any lollies the first day, and need to find the maximum number of lollies we can get during days 2 through n. Overall, the maximum number of lollies we can get during days 1 through n is the maximum of the two possibilities.

Applying this idea recursively leads to subproblems of the following form: For $1 \le i \le n$, $\mathrm{OPT}(i)$ denotes the maximum number of lollies we can get during days i through n. The above discussion yields the following recurrence:

$$\mathrm{OPT}(i) = \max\left\{\ell_i + \mathrm{OPT}(i + k_i + 1), \mathrm{OPT}(i+1)\right\},$$

where $\mathrm{OPT}(i) = 0$ for $i > n$. We use the recurrence to compute $\mathrm{OPT}(i)$ for $i = n, n-1, \ldots, 1$, and return $\mathrm{OPT}(1)$.

There are n subproblems, and each update takes time $O(1)$. Therefore the total running time is $O(n)$.

Alternate solution. As an alternate solution, we can efficiently reduce this problem to weighted interval scheduling. There is one interval for every day i, $1 \leq i \leq n$. The interval corresponding to day i is $[i, i + k_i]$ and has weight ℓ_i. With this setup, a valid selection of days on which we get lollies corresponds to a valid interval schedule, and vice versa. Moreover, the total number of lollies we get equals the weight of the intervals scheduled.

Note that, apart from the initial sorting phase and the construction of the table p of predecessors, the weighted interval scheduling algorithm from class takes time $O(n)$. Also, while we sorted the intervals by nondecreasing end time and went over them from back to front, we could as well sort them by nondecreasing start time and go over them from the front to the back. (This is like reverting the direction of the time axis.) Since we are given the intervals sorted by their start time, we do the latter as it obviates the need for the initial sorting. Moreover, we have that $p(i) = i + k_i + 1$, so the table p can be computed in time $O(n)$. With these provisos, the alternate solution also runs in $O(n)$ time.