# Algorithms and Data Structues 2020/21

## Coursework 2 — Solution

This coursework is due by **4:00pm, on Friday, 20. November 2020**. This is a firm deadline. Solutions must be submitted by uploading them on the LEARN page (www.learn.ed.ac.uk) of ADS 2020/21, Left Menu item "Assessment", item on page "Coursework 2". Multiple submissions are possible, but only the last submission counts.

This coursework 2 is **summative**. It counts for 50% of the overall course grade.

**Late policy:** http://www.inf.ed.ac.uk/student-services/teaching-organisation/for-taught-students/coursework-and-projects/late-coursework-submission

**Conduct policy:** http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

1. Consider the problem of taking a set of $n$ items with sizes $s_1, \ldots, s_n$, and values $v_1, \ldots, v_n$ respectively. We assume $s_i, v_i \in \mathbb{N}$ for all $1 \leq i \leq n$. Suppose we are also given a capacity $C \in \mathbb{N}$.

   The packing problem is the problem of finding a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} s_i \leq C$ and such that $\sum_{i \in S} v_i$ is maximized subject to the first constraint.

   We write $P_{n,C}$ to denote the value $\sum_{i \in S} v_i$ of the maximum-value packing on the set of all items. For any $k \leq n$, and any $\widehat{C} \leq C, \widehat{C} \in \mathbb{N}$, we can consider the same problem on the first $k$ items in regard to capacity $\widehat{C}$. We denote the maximum-value packing for such a subproblem by $P_{k,\widehat{C}}$.

   The goal is to develop a $\Theta(n \cdot C)$ dynamic programming algorithm to compute the optimal packing wrt. the original $n$ items and capacity $C$.

   (a) Prove a suitable recurrence for $P_{k,\widehat{C}}$ that holds for all $k \leq n$ and $\widehat{C} \leq C$. [*25 marks*]

   **Answer:** We prove that the following recurrence holds:

$$P_{k,\widehat{C}} = \begin{cases} 0 & \text{if } k = 0 \\ P_{k-1,\widehat{C}} & \text{if } k > 0 \text{ but } s_k > \widehat{C} \\ \max\{P_{k-1,\widehat{C}}, P_{k-1,\widehat{C}-s_k} + v_k\} & \text{otherwise.} \end{cases}$$

   (i) The first case, when $k = 0$ is obvious. We have no items to pack, so the optimal value is 0.

   If $k \geq 1$, then we focus on the final item in $\{1, \ldots, k\}$. This have value $v_i$ and size $s_i$.

   (ii) In the case that $s_k > \widehat{C}$, no feasible packing for $k, \widehat{C}$ can contain item $k$. The optimal solution is the same as the optimal one on the first $k - 1$ items with the same capacity bound.

(iii) In the case that $s_k \leq \widehat{C}$, the set of feasible packings can be partitioned depending on whether they contain item $k$, or do not contain item $k$.

By definition,

$$P_{k,\widehat{C}} \;=\; \max_{S\subseteq\{1,\ldots,k\}} \left\{ \sum_{i\in S} v_i \;:\; \sum_{i\in S} s_i \leq \widehat{C} \right\}.$$

The set of all $S$ to be considered can be partitioned according to whether $k \in S$ or $k \notin S$. Using this partitioning, we can rewrite $P_{k,\widehat{C}}$ as

$$\max \left\{ \max_{S\subseteq\{1,\ldots,k-1\}} \left\{ \sum_{i\in S} v_i \;:\; \sum_{i\in S} s_i \leq \widehat{C} \right\}, \max_{S\subseteq\{1,\ldots,k-1\}} \left\{ v_k + \sum_{i\in S} v_i \;:\; s_k + \sum_{i\in S} s_i \leq \widehat{C} \right\} \right\},$$

where the left internal $\max$ selects the optimum knapsack not containing item $k$, and the right internal $\max$ selects the optimum knapsack that does contain item $k$. Now observe that by definition of $P_{k-1,\widehat{C}}$, this implies

$$P_{k,\widehat{C}} \;=\; \max \left\{ P_{k-1,\widehat{C}}, \max_{S\subseteq\{1,\ldots,k-1\}} \left\{ v_k + \sum_{i\in S} v_i \;:\; s_k + \sum_{i\in S} s_i \leq \widehat{C} \right\} \right\}.$$

Also note that we have $s_k + \sum_{i\in S} s_i \leq \widehat{C}$ if and only if $\sum_{i\in S} s_i \leq \widehat{C} - s_k$, hence we have

$$\begin{aligned} P_{k,\widehat{C}} &= \max \left\{ P_{k-1,\widehat{C}}, \max_{S\subseteq\{1,\ldots,k-1\}} \left\{ v_k + \sum_{i\in S} v_i \;:\; \sum_{i\in S} s_i \leq \widehat{C} - s_k \right\} \right\} \\ &= \max \left\{ P_{k-1,\widehat{C}},\; v_k + \max_{S\subseteq\{1,\ldots,k-1\}} \left\{ \sum_{i\in S} v_i \;:\; \sum_{i\in S} s_i \leq \widehat{C} - s_k \right\} \right\} \\ &= \max \left\{ P_{k-1,\widehat{C}},\; v_k + P_{k-1,\widehat{C}-s_k} \right\}, \end{aligned}$$

where the final step follows by definition of $P_{k-1,\widehat{C}-s_k}$.

(b) Use your recurrence above to develop a $\Theta(n \cdot C)$ dynamic programming algorithm to compute the optimal packing wrt. the original $n$ items and capacity $C$. Formally justify the $\Theta(n \cdot C)$ runtime of your algorithm. [*25 marks*]

**Answer:** The main issues to be considered in solving are dp1(a) and dp1(b) (the collection of subproblems and the recurrence relating the problems), dp2 (the table(s) where the results will be stored) and dp3 (the order of filling in the table(s)). For dp3, the order of filling in the table has to ensure the subproblems on the rhs of the recurrence have *always* been solved and stored (hence available for lookup) in advance of the problem on the lhs.

Now we give our solution:

dp1 dp1(a) and dp1(b). These decisions are easily made by reference the recurrence above in 2(a). This recurrence contains $P_{k,C'}$ terms on the right-hand side, for what seems like fairly changeable values of $C' \leq C$. Hence we will decide to solve $P_{k,C'}$ for all $0 \leq k \leq n$ and all $C' \in \mathbb{N}, C' \leq C$.

dp2 We define two tables of size $(n+1) \cdot (C+1)$ each, one called $P$, the other called $s$. The $P$ table stores integers (the values of the "best" knapsacks) and the $s$ table stores binary values. For any $0 \leq j \leq n$, $0 \leq \widehat{C} \leq C$, the entry $P[j, \widehat{C}]$ will denote the value of the best knapsack solution from items $1, \ldots, j$ wrt capacity $\widehat{C}$ - that is the value of $P_{j,\widehat{C}}$. The auxiliary table $s$ is defined as follows - $s[j, \widehat{C}]$ will be 1 if an optimal solution *does* include item $j$ and 0 otherwise.

Note that the space used by our algorithm is already $\Theta(n \cdot C)$.

dp3 The tables are filled in increasing order of $j$, and then in increasing order of $\widehat{C}$. Initialize the 0th row and 0th column of $P$ and of $s$ to contain all 0s. Note that this initialization of the 0th row takes care of all instances of the "first case" of our recurrence.

Next we consider each $j$ from $1, \ldots, n$ in turn, and for a particular $j$ also consider all $\widehat{C}$s in increasing order. For a specific $j$, $\widehat{C}$, test whether $s_i \leq \widehat{C}$ (this takes $\Theta(1)$ time), and depending on the result, either do a lookup of $P[j-1, \widehat{C}]$ or of both $P[j-1, \widehat{C}]$ and also $P[j-1, \widehat{C} - s_i]$. Note that by $j - 1 < j$, we have *previously* visited these cells and filled them, hence these lookups are immediate, taking $\Theta(1)$ time. Then, with these values, compare $P[j-1, \widehat{C}]$ with $P[j-1, \widehat{C} - s_i] + v_i$ ($\Theta(1)$ time). Take the maximum and assign $P[j, \widehat{C}]$ this value. If the first is larger, set $s[j, \widehat{C}]$ to be 0, if the second is larger, set $s[j, \widehat{C}]$ to be 1.

The two tables can be entirely completed in $\Theta(n \cdot C)$ time. To find the actual knapsack solution (rather than just its value), we finish by starting with $j = n, \widehat{C} = C$, and outputting $'j,'$ if and only if $s[j, \widehat{C}] = 1$, then recursing either on cell $[j-1, \widehat{C}]$ or $[j-1, \widehat{C} - s_i]$.

2. King Arthur has problems to administer his realm. His court contains $n$ knights and he rules over $m$ counties. The knights differ in their abilities and local popularity: Each knight $i$ can oversee at most $q_i$ counties, and each county $j$ will revolt unless it is overseen by some knight in a given subset $S_j \subseteq \{1, \ldots, n\}$ of the knights. Only 1 knight can oversee a county, to prevent conflicts between the knights.

   The king discusses the problem with his court magician Merlin.

   (a) Show how Merlin can use the Max-Flow algorithm to efficiently compute an assignment of the counties to the knights that prevents a revolt, provided that one exists.

   How can he determine whether the algorithm was successful?

   Prove the complexity of this algorithm, *in terms of some function* $F(v, e)$, where $F(v, e)$ denotes the running time of a Max-Flow algorithm on a graph with $v$ vertices and $e$ edges. [*30 marks*]

   **Answer (10+10+10 for construction+correctness+runtime):** Merlin constructs a graph with $n + m + 2$ vertices, $n$ vertices $k_1, \ldots, k_n$ corresponding to the knights, $m$ vertices $c_1, \ldots, c_m$ for the counties and two special vertices $s$ and $t$. He puts an edge from $s$ to $k_i$ with capacity $q_i$. Moreover, he puts an edge with capacity 1 from $k_i$ to $c_j$ iff $i \in S_j$ (i.e., iff county $j$ may be ruled by knight $i$). Finally, put an edge with capacity 1 from $c_j$ to $t$. Now run the Max-Flow algorithm to find a maximal flow in the graph. The algorithm is successful iff this flow has value $m$.

   Why is this algorithm correct? First, we show that a max flow of $m$ yields a complete allocation of the knights to the counties. The max flow is integral and has value $m$. Thus each of the $m$ vertices $c_j$ has outflow 1 and an incoming flow 1 from exactly one vertex $k_i$. Thus county $j$ is overseen by knight $i$ where $i \in S_j$. Moreover, since the flow satisfies the capacity constraints in the flow graph, no knight oversees more counties than his capacity. Secondly, a complete allocation of the knights to the counties satisfies the constraints and yields a flow of value $m$.

   Runtime of the algorithm: The constructed graph has $n + m + 2$ vertices and $n + m + \sum_j |S_j|$ edges. Thus it takes $\Theta(n + m + \sum_j |S_j|)$ time to construct. Running the Max-Flow algorithm on it takes $F(n + m + 2, n + m + \sum_j |S_j|)$. So the total runtime of the algorithm is $\Theta(n + m + \sum_j |S_j|) + F(n + m + 2, n + m + \sum_j |S_j|)$.

   (b) Suppose that Merlin runs the Max-Flow algorithm on the encoded instance, but it does not produce a solution that fits the constraints and prevents all counties from revolting.

   King Arthur demands an explanation. While he believes that the algorithm/encoding (and proof) is correct, he doubts that Merlin has executed the algorithm correctly on the given large instance. Merlin needs to convince the king that no suitable assignment is possible under the given constraints. Re-running the algorithm step-by-step in front of the king is not an option, because the king does not have that much time.

   How can Merlin *quickly* convince the king that there is no solution, based on the structure of the Max-Flow problem? (Note that the argument must work for every instance where there is no solution, not just a particular instance.) [*20 marks*]

   **Answer:** If there is no flow of size $m$ then there must be a cut in the graph with a capacity $< m$. From the structure of this cut we obtain a set of counties $T \subseteq$

$\{1, \ldots, m\}$ such that if $U$ is the union of the sets $S_j$ for $j \in T$, then $\sum_{i \in U} q_i < |T|$. I.e., there is a subset of counties $T$ such that the combined capacity of the knights that they are willing to be overseen by is smaller than the number of counties in that subset.

Merlin presents the sets $T$ and $U$ to the king to explain why no solution exists. (I.e., at least one county in $T$ will revolt, regardless of how the knights are assigned.)