Analysis of Algorithms: Solutions 9

The histogram shows the distribution of grades for the homeworks submitted on time.

Problem 1
Write pseudocode of an algorithm GREEDY-KNAPSACK(W, v, w, n) for the 0-1 Knapsack Problem, and give its running time. The arguments are an weight limit W, array of item values v[1..n], and array of item weights w[1..n]. Your algorithm must use the greedy strategy described in class, and return the set of selected items.

GREEDY-KNAPSACK(W, v, w, n)

sort items in the descending order of the \( \frac{v[i]}{w[i]} \) ratios

\( \text{items} \leftarrow \emptyset \) \> Set of selected items.
\( \text{w-sum} \leftarrow 0 \) \> Sum of their weights.

\( \text{for } i \leftarrow 1 \text{ to } n \) \> In sorted order.
  \( \text{do if } \text{w-sum} + w[i] \leq W \)
    \( \text{then } \text{items} \leftarrow \text{items} \cup \{i\} \)
    \( \text{w-sum} \leftarrow \text{w-sum} + w[i] \)

\text{return } \text{items}

The sorting takes \( O(n \lg n) \) time, whereas the selection loop runs in linear time. Thus, the complexity of \text{GREEDY-KNAPSACK} is \( O(n \lg n) \).
Problem 2
Using Figure 17.4(b) in the textbook as a model, draw an optimal-code tree for the following set of characters and their frequencies:

\[
\begin{align*}
&\text{a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21}
\end{align*}
\]

![Optimal Code Tree](image)

Problem 3
Suppose that you drive along some road, and you need to reach its end. Initially, you have a full tank, which holds enough gas to cover a certain distance \( d \). The road has \( n \) gas stations, where you can refill your tank. The distances between gas stations are represented by an array \( A[1..n] \), and the last gas station is located exactly at the end of the road. You wish to make as few stops as possible along the way. Give an algorithm \textsc{Choose-Stops}(d, A, n) that identifies all places where you have to refuel, and returns the set of selected gas stations.

\textsc{Choose-Stops}(d, A, n)

\begin{align*}
\text{stations} &\leftarrow \emptyset \quad \triangleright \text{Set of selected gas stations.} \\
\text{d-left} &\leftarrow d \quad \triangleright \text{Distance that corresponds to the remaining gas.} \\
\text{for } i &\leftarrow 1 \text{ to } n \\
&\quad \text{do if } d-left < A[i] \quad \triangleright \text{Cannot reach the next gas station? Then refuel.} \\
&\hspace{1cm} \text{then stations} \leftarrow \text{stations} \cup \{i - 1\} \\
&\hspace{1cm} \text{d-left} \leftarrow d \\
&\hspace{1cm} \text{d-left} \leftarrow d-left - A[i] \quad \triangleright \text{Drive to the next station.} \\
\text{return stations}
\end{align*}

The algorithm runs in linear time, that is, its complexity is \( \Theta(n) \).
Problem 4
Suppose that the weights of all items in the 0-1 Knapsack Problem are integers, and the weight limit $W$ is also an integer. Design an algorithm that finds a globally optimal solution, and give its time complexity in terms of the number of items $n$ and weight limit $W$.

We use dynamic programming with two arrays, $\textit{item}[1..W]$ and $\textit{value}[0..W]$, which are indexed on the size of a knapsack. For every size $i$ between 0 and $W$, we compute the maximal value of items that can be loaded into a knapsack, and store this result in $\textit{value}[i]$. If $\textit{value}[i]$ is larger than $\textit{value}[i-1]$, then $\textit{item}[i]$ is the last added item; otherwise, $\textit{item}[i]$ is 0.

We add items in their numerical order; that is, if items $j_1$ and $j_2$ must be in the knapsack, and $j_1 < j_2$, then we add $j_1$ before $j_2$.

The following algorithm computes the arrays $\textit{item}[1..W]$ and $\textit{value}[0..W]$, and returns the maximal value of items for size $W$; its time complexity is $\Theta(nW)$.

**Dynamic-Knapsack**($W, v, w, n$)

\[ \textit{value}[0] \leftarrow 0 \]

for $i \leftarrow 1$ to $W$  // Consider every size of a knapsack.

\[ \textit{do item}[i] \leftarrow 0 \]

\[ \textit{value}[i] \leftarrow \textit{value}[i-1] \]  // Initialize the maximal value for size $i$.

for $j \leftarrow 1$ to $n$  // Look through items, to find the best addition to a smaller load.

\[ \textit{do if w}[j] \leq i \]  // Item $j$ fits into the knapsack.

\[ \textit{and j > item}[i - \textit{w}[j]] \]  // It does not violated the numerical order.

\[ \textit{and value}[i] < \textit{value}[i - \textit{w}[j]] + v[j] \]  // We get a good value by adding $j$.

\[ \textit{then item}[i] \leftarrow j \]  // Add $j$ to the knapsack.

\[ \textit{value}[i] \leftarrow \textit{value}[i - \textit{w}[j]] + v[j] \]

return $\textit{value}[W]$

We also need an algorithm for printing out the list of selected items. The following output procedure uses the array $\textit{item}[1..W]$, built by **Dynamic-Knapsack**, to print items in their numerical order; its running time is $O(n)$.

**Print-Knapsack**($\textit{item}, W, w, i$)

\[ \textit{if} i = 0 \]

\[ \textit{then} \text{“do nothing”} \]

\[ \textit{elseif item}[i] = 0 \]

\[ \textit{then Print-Knapsack}(\textit{item}, W, w, i - 1) \]

\[ \textit{else Print-Knapsack}(\textit{item}, W, w, i - \textit{w}[item[i]]) \]

\[ \text{print item[i]} \]