Analysis of Algorithms: Solutions 9

```
                                       X
                                       X
                                       X
                                       X
                                       X
                                       X
                                       X
                                       X
number of                              X
homeworks                              X
                           X           X
                           X           X
                           X           X
                     X     X           X
            X        X     X     X  X  X
            X  X  X  X  X  X  X  X  X  X
            -----------------------------
            1  2  3  4  5  6  7  8  9  10
                        grades
```

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

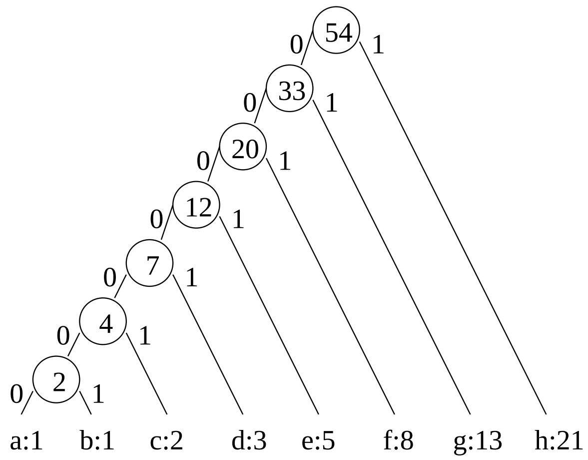
Problem 1
Write pseudocode of an algorithm GREEDY-KNAPSACK(W, v, w, n) for the 0-1 Knapsack Problem, and give its running time. The arguments are an weight limit W, array of item values $v[1..n]$, and array of item weights $w[1..n]$. Your algorithm must use the greedy strategy described in class, and 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
items $\leftarrow \emptyset$ $\quad\triangleright$ Set of selected items.
w-sum $\leftarrow 0$ $\quad\triangleright$ Sum of their weights.
for $i \leftarrow 1$ **to** n $\quad\triangleright$ In sorted order.
  do if *w-sum* $+ w[i] \leq W$
    then *items* $\leftarrow$ *items* $\cup \{i\}$
      w-sum $\leftarrow$ *w-sum* $+ w[i]$
return *items*

The sorting takes $O(n \lg n)$ time, whereas the selection loop runs in linear time. Thus, the complexity of 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:

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21



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 Choose-Stops(d, A, n) that identifies all places where you have to refuel, and returns the set of selected gas stations.

Choose-Stops(d, A, n)
stations $\leftarrow \emptyset$ $\triangleright$ Set of selected gas stations.
d-left $\leftarrow d$ $\triangleright$ Distance that corresponds to the remaining gas.
for $i \leftarrow 1$ **to** n
  do if *d-left* $< A[i]$ $\triangleright$ Cannot reach the next gas station? Then refuel.
    then *stations* $\leftarrow$ *stations* $\cup \{i-1\}$
      d-left $\leftarrow d$
  d-left $\leftarrow$ *d-left* $- A[i]$ $\triangleright$ Drive to the next station.
return *stations*

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, $item[1..W]$ and $value[0..W]$, 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 $value[i]$. If $value[i]$ is larger than $value[i-1]$, then $item[i]$ is the last added item; otherwise, $item[i]$ 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 $item[1..W]$ and $value[0..W]$, and returns the maximal value of items for size W; its time complexity is $\Theta(nW)$.

```
DYNAMIC-KNAPSACK(W, v, w, n)
value[0] ← 0
for i ← 1 to W    ▷ Consider every size of a knapsack.
    do item[i] ← 0
       value[i] ← value[i − 1]    ▷ Initialize the maximal value for size i.
       for j ← 1 to n    ▷ Look through items, to find the best addition to a smaller load.
          do if w[j] ≤ i    ▷ Item j fits into the knapsack.
                    and j > item[i − w[j]]    ▷ It does not violated the numerical order.
                    and value[i] < value[i−w[j]]+v[j]    ▷ We get a good value by adding j.
                then item[i] ← j    ▷ Add j to the knapsack.
                     value[i] ← value[i − w[j]] + v[j]
return value[W]
```

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

```
PRINT-KNAPSACK(item, W, w, i)
if i = 0
   then "do nothing"
elseif item[i] = 0
   then PRINT-KNAPSACK(item, W, w, i − 1)
else PRINT-KNAPSACK(item, W, w, i − w[item[i]])
     print item[i]
```