Nicholas Gin
300107597

CSI 2120 Comprehensive Assignment – Part 4: Scheme Document

Contents

Nicholas Gin
300107597

Function Descriptions

These same function descriptions in the actual context of the code, as well as any additional details concerning how they work, can be found in the comments of *solveKnapsack.rkt*.

Each function's name and parameters are provided in this document for reference.

(define (solveKnapsack filename)
The function that is called by the user; it calls all other functions to solve the knapsack problem and to produce a solution file. solveKnapsack takes a filename as a parameter (the name of the file that will be read for the information about the items being considered in the knapsack problem). The following are the steps solveKnapsack takes to produce the solution file. Get the maxWeight of the knapsack from the file by calling the getMaxWeight function; bind it to maxWeight. Translate the contents of the file into a list; bind this list to fileContents. For example, the contents of p1.txt will be translated into the following list: '(4 A 1 1 B 6 2 C 10 3 D 15 5 7). Determine the length of fileContents but not including the first and last elements of this list (the first and last elements of this list represent the number of items being considered and the maximum capacity of the knapsack respectively) by finding the length of fileContents and subtracting 2; bind this difference to size. Use splitUp to create a list containing sublists that represent each item as they are described in the file; bind this list to items. Since we only want to consider the information of the items in fileContents and not the number of items being considered and the maximum capacity of the knapsack, (the first and last elements in fileContents respectively) we do not call splitUp on the entirety of the fileContents list. We remove the last element of fileContents by calling removeLast on it and we remove its first element by calling cdr on it. This is the list we call splitUp on. Again, the length of this list is indicated by the value we bound to "size". Provide an empty list for the parameterized newLst of splitUp, as we want to create a list containing all of the sublists from scratch. For more information on how the splitUp function works, please see the splitUp function. Call knapsack (with maxWeight and items as parameters) to produce the filled knapsack (the solution) for the specified items and knapsack capacity as detailed in the file given by the user; bind the solution to solution. The solution produced by knapsack is a list that contains two elements: the total value of the solution knapsack and a sublist containing the names of all the elements in the solution knapsack. For example, the solution of p1.txt is '(21 (B D)). Because of the way the function createSolution works, we want to take the item names out of the sublist and put them into the main list with the total value; in other words, we want to convert the solution list into '(21 B D). We do this by calling append on the (car (cdr solution)) (the sublist of items) and (car solution) (the total value). We call createSolution on the list we just created as well as the filename of the input file provided by the user. createSolution will produce a solution file, named the given filename.sol, that displays the contents of the solution knapsack.

(define (translateFileToList filename output)
This function is taken and slightly adapted from Slide 12 of Scheme4_IO_sort slides. This function translates all contents of the user-provided file into a list. For example, the contents of p1.txt will be translated into a list: '(4 A 1 1 B 6 2 C 10 3 D 15 5 7). The following are the steps

this function takes. Open the file; assign this to p. Read from the file p and assign what we read from it to x. If x is the end of the file, close the file and return an empty list. Else, add whatever we read to whatever we are going to read from the recursive call to the file (we are constructing a list of all the contents in the file).

(define (splitUp lngth lst newLst)
This function splits up the list containing the contents of the user-provided file into a new list containing sublists that represent each item. In order to carry out this process, this function takes the following as parameters: the length of the list we are splitting up (lngth), the list we are splitting up (lst), and the list we form that contains all of the sublists (newLst). In the case of p1.txt, via solveKnapsack, we will pass the following list, lst, to be split up into sublists: '(A 1 1 B 6 2 C 10 3 D 15 5 7). For every 3 elements in lst, the function will create a sublist that is added to newLst; each sublist represents all the details about an item. So, in this case, newLst will be: '((A 1 1) (B 6 2) (C 10 3) (D 15 5)). The following are the steps this function takes. If lst is empty, we have iterated through the entirety of lst and we have finished creating newLst. If lngth > 1, call "take" on lst for the first 3 elements of lst. For example, the first iteration of this function for lst, '((A 1 1) (B 6 2) (C 10 3) (D 15 5)), will produce the sublist '(A 1 1). We then add this sublist to the newList. Decrease lngth by 3 and remove the first three elements of lst by calling cdr three times on it. We then recursively call this function for the rest of this modified, shortened lst, repeating the same steps until lst is empty.

(define (removeLast lst)
This function removes the last element of a list, lst. To accomplish the removal of the last element of lst, we first reverse the ordering of lst. We then remove the first element from the reversed lst by calling cdr, and finally reverse the reversed lst to restore the original ordering of lst (but without the last item in lst present).

(define (getMaxWeight filename)
This function retrieves the maximum weight/capacity that the knapsack can hold given an input file. The following are the steps this function takes. Convert the contents of the file into a list using the translateFileToList function; bind this list to lst. The maximum weight that the knapsack can hold will always be indicated on the last line of the input file provided by the user. This means that the maximum weight can be found as the last element of lst, return the last element of lst as the maximum weight.

(define (itemValue itemSublist)
This function gets the value associated with an item. From an item sublist, the second value in the sublist indicates the value of the item. For example, in the item sublist, '(B 6 2), 6 is the item's value. To accomplish its goal, the function gets the second value from the parameterized itemSublist by calling cadr on the itemSublist.

(define (itemWeight itemSublist)
This function gets the weight associated with an item. From an item sublist, the third value in the sublist indicates the weight of the item. For example, in the item sublist, '(B 6 2), 2 is the item's weight. To accomplish its goal, the function gets the third value from the parameterized itemSublist by calling caddr on the itemSublist.

(define (sumTotalKnapsackValue knapsack)
This function gets the culminative, total value of all the items in a parameterized knapsack. Use map to create a list containing the itemValues of each item in the parameterized knapsack. For example, mapping itemValue to the knapsack represented by the list '((A 1 1) (B 6 2) (C 10 3) (D 15 5)) will produce the list '(1 6 10 15), which is the list containing each item from the knapsack's itemValue (itemValue is called on each sublist in the list). Use apply + on the mapping to sum together all of the itemValues in that list. This sum represents the culminative, total value of all the items in the knapsack.

(define (sumTotalKnapsackWeight knapsack)
This function gets the culminative, total weight of all the items in a parameterized knapsack. Use map to create a list containing the itemWeights of each item in the parameterized knapsack. For example, mapping itemWeight to the knapsack represented by the list '((A 1 1) (B 6 2) (C 10 3) (D 15 5)) will produce the list '(1 2 3 5), which is the list containing each item from the knapsack's itemWeight (itemWeight is called on each sublist in the list). Use apply + on the mapping to sum together all of the itemWeights in that list. This sum represents the culminative, total weight of all the items in the knapsack.

(define (max knapsackWithItem knapsackWithoutItem maxWeight)
This function compares the totalValues of two knapsacks: a knapsack with an item included (knapsackWithItem) and a knapsack without an item included (knapsackWithoutItem). If the total weight of the knapsackWithItem is not greater than the maxWeight (provided in parameters) that the knapsack can contain and the total value of the knapsackWithItem is greater than the total value of the knapsackWithoutItem, then return the knapsackWithItem. Else, return the knapsackWithoutItem. Of the two given knapsacks, we want to return the knapsack with the highest total value. To determine the total weight of knapsackWithItem, we call sumTotalKnapsackWeight on it. To determine the total values of the two knapsacks, we call sumTotalKnapsackValues on each of these knapsacks respectively.

(define (knapsack maxWeight items)
This function is included to match the function described in the assignment outline.
It takes, as parameters, a maximum weight that the knapsack can hold (maxWeight) as well as a list containing sublists that represent the items that can be put into the knapsack (items). The function then calls bruteForce using the parameterized maxWeight and items to produce a filledKnapsack (the solution to the knapsack problem); the filledKnapsack is first set to an empty list since items will be added to it. To output the information about the solution knapsack in the format described in the assignment outline, we create a list that contains the total value of the

solution knapsack (to calculate this value, we call sumTotalKnapsackValue on the solution knapsack) and a sublist containing the names of all the items in the solution knapsack (to create the sublist, we map the first element in every sublist in the solution knapsack to a new list and reverse the ordering of this list so the names are in alphabetical order).

(define (bruteForce maxWeight filledKnapsack items)
This function performs the brute force algorithm to solve the knapsack problem for a given list that contains sublists which represent the items that can be put into the knapsack (items) and a maximum weight that the knapsack can contain (maxWeight). The final solution, the set of items that maximizes the total value of the knapsack for its maximum weight, will be placed in filledKnapsack (the solution). For each item described in "items", we make a binary decision: should we add the item to the filledKnapsack or not? This binary decision is represented by the two branches of a binary tree. We will inspect the total value of the filledKnapsack twice: after adding the item to it and after not adding the item to it. We will return the filledKnapsack in the state where it will have a higher total value, either with the item added to it or not via a recursive call to this function and a call to the max function. For each recursive call to this function, we will consider a different item from "items." The following are the steps this function takes. If there are no items to put into the filledKnapsack, then return filledKnapsack. If there are items to consider, then we consider two cases: filledKnapsack after adding an item to it (by adding the first found element in "items" (car items) to the filledKnapsack) and filledKnapsack without an item added to it. Calling the function, max, providing both of these cases as separate parameterized knapsacks will return the knapsack with the higher total value between the two. For the first case, recursively call this function after adding the item to the Knapsack; consider the next items in "items" in this function call by considering "items" with its first element removed (call cdr on items). For the second case, recursively call this function without adding the item to the Knapsack; consider the next items in "items" in this function call by considering "items" with its first element removed (call cdr on items). After the algorithm completes and the binary tree is finished, we find filledKnapsack with the combination of items that maximizes the knapsack's total value without exceeding its max weight capacity.

(define (createSolution lst filename)
The function used to create the solution file. The function takes a filename as input, so that the solution file can be written in the form filename.sol. This function calls an adapted version of proc-out-file from Slides 13-14 of Scheme4_IO_sort slides to write the solution file. The message, “Solution file created,” will be displayed to the user once the file has been created. More details on how this function works are found in the comments of *solveKnapsack.rkt*.

References

Knapsack Problem/0-1. (2020, December 21). Rosetta Code. Retrieved April 4, 2021, from https://rosettacode.org/wiki/Knapsack_problem/0-1#Racket.

Scheme4_IO_sort. (n.d.). Brightspace. Retrieved April 4, 2021, from https://uottawa.brightspace.com/d2l/le/content/213483/viewContent/3605436/View.