

Homework 8 Solutions

1)

The goal is to fill the knapsack and maximize the total sum of values of the items packed. Because fractions of items can be added, this problem can be solved with a greedy approach. The basic idea is to balance the value of the item added with the size of the item. The more valuable an item is, the more desirable it will be to add to the knapsack. The value of the item can be computed as the value of the item divided by the size of the item. To solve the fractional knapsack problem, simply calculate the value of each item, sort the items by the value, and then add all of the most valuable, followed by the second most, and so forth. At some point, either the knapsack will be full, in which case the algorithm is done, or there will be no further items to add, or there will be more of some item than there is space in the knapsack. Add the amount of this final item to the knapsack required to fill the knapsack. This solves the fractional knapsack problem with an $O(n)$ scan to calculate the values, an $O(n \log n)$ sort, and an $O(n)$ sort giving $O(n \log n)$ time.

Algorithm 1 [T] =fracKnapsack(K, S, V)

1: **Input:** K (the size of the knapsack), S (an array of size n storing both the sizes and the values of elements being used to fill the knapsack)

2: **Output:** A (a list of items containing the elements used to fill knapsack or a statement stating that the knapsack can not be filled)

3: **begin**

4: /* initialize the densities of each element to 0*/

5: **for** all elements e **do**

6: $e.density = 0$;

7: **end for**

8: /* calculate and store densities of each element */

9: **for** all elements e **do**

10: $e.density = e.value/e.size$;

11: **end for**

12: /*Sort the elements with respect to their density in descending order*/

13: /*you are able to use any fast sorting algorithms you know such as quicksort, or even store them in a heap etc...*/

14: /*Initialize packed to 0, this variable represents how much of the knapsack is already packed*/

15: $packed = 0$;

16: /*Start from beginning of list, element should contain largest density*/

17: **while** $packed < K$ and there are elements still in the list **do**

18: **if** $e.size < K - packed$ **then**

19: /*add element to the list containing the elements filling knapsack;*/

20: $packed = packed + e.size$;

21: **else**

22: /*calculate portion still needed to fill*/

23: $portion = K - packed$;

24: /*change elements size to represent the size of element which as used*/

25: $e.size = portion$;

26: /*add element to the list containing the elements filling the knapsack;*/

27: $packed = packed + e.size$;

28: **end if**

29: **end while**

30: **if** $packed = K$ **then**

31: return A ;

32: **else**

33: /*Knapsack could not be filled;*/34: **end if**35: **end**

2)

For this problem, we are asked to prove that arranging a set of points into the standard polygon representation takes $\Omega(n \log n)$ time. This proof will be by contradiction.

Assume that a set of points can be put into the standard representation in less than $\Omega(n \log n)$ time. Specifically, the set of points can be put into the standard representation in $O(f(n))$ time.

Consider a set of real numbers a_1, a_2, \dots, a_n .

Create a set of points by applying $y=x^2$ to each of these points, giving a set of points $(a_1, a_1^2), (a_2, a_2^2), \dots, (a_n, a_n^2)$. This takes $O(n)$ time.

Apply the algorithm to put the points into standard polygon order.

These points are arranged on a parabola, a convex polygon. Note that because $y=x^2$ is a function, there is no x coordinate that is defined at more than one y coordinate. Because of this, the standard polygon order will return the points in order of increasing x coordinate. Scan the list and return the sorted x coordinates, giving the set of real numbers in sorted order. Sorting a set of arbitrary real numbers takes $O(n + f(n))$ time using this method. If $f(n) < n \log n$ then an arbitrary set of real numbers can be sorted in less than $O(n \log n)$ time, which contradicts a known fact that sorting is $\Omega(n \log n)$.

Therefore, arranging a set of points into the standard polygon representation takes $\Omega(n \log n)$ time.

3)

To find a solution for the partition problem in $O(nS)$ time, note that the partition problem can be mapped to the knapsack problem in a very straightforward manner. Take an $O(n)$ scan of the items to be partitioned, and sum them up to find S . Then, run the standard knapsack problem with a knapsack size of $S/2$ (if S is not even, there is no solution). Run the algorithm exactly as it is in 5.10 on page 110 of Manber. This takes $O(nK)$ time, and in this case $K=S/2$. So this is $O(nS/2)=O(nS)$ time, with an $O(n)$ scan solves the partition problem in $O(nS)$ time.

To show that the partition problem is in NP, a non deterministic, polynomial time algorithm must be found. Create two empty sets. Make an $O(n)$ scan of the list of items, and for each item non-deterministically place each item in one set or the other. Take an $O(n)$ scan of each set when all items are exhausted, and determine if they are of the same size. If they are then there is a partition.

4)

- 1) Show that the constrained path problem is in NP.

Start at s . Non-deterministically select an unmarked vertex from the vertices adjacent to s that is not already added and mark it. Add its weight to a counter. Do the same thing on the new vertex. Repeat this until the weight exceeds L , there are no vertices left to add to the path, or t is added.

- 2) Show that the constrained path problem is NP Complete.

In order to show that constrained path is NP Complete, the 0-1 knapsack problem can be mapped to the constrained path problem.

Consider the knapsack problem with a knapsack of size K , and n items. Each item has size s_1, s_2, \dots, s_n .

The goal is to represent the knapsack problem as a graph. The structure of the graph should be such that traversing an edge is equivalent to adding an item to the knapsack.

Create a weighted, directed graph with $n+2$ vertices. Create a source vertex s , a sink vertex t , and one node v_0, v_1, \dots, v_n for each item in the knapsack. Create an edge from s to each vertex in the graph, except for t , with a cost of 0. For each vertex v_i other than s and t , create an edge from v_i to every other vertex v_{i+1}, \dots, v_n and to t with weight s_i . Do not create "back" edges to v_0, v_1, \dots, v_{i-1} because these edges can allow items to be added more than once.

This allows the knapsack problem to be modeled in a graph. Consider a path through the graph. Whenever this path traverses an edge, this is equivalent to adding an item to the knapsack. The node s represents an empty knapsack, which can add any item. Once the path exits a node, its weight is added to the current path cost. Once the path enters t , there are no out edges and this is equivalent to finishing.

This transformation takes $O(n^2)$ time, creating $O(n)$ vertices with edges to each of $O(n)$ vertices.

Run the constrained path algorithm on this graph, with $L=K$ (the knapsack size). If a path exists in the graph, then a solution for the knapsack problem exists. Therefore, the knapsack problem can be solved in $O(n^2 + f(n))$ time, where $f(n)$ is the running time of the constrained path algorithm. So, if the constrained path problem can be solved in polynomial time, then the knapsack problem can be solved in polynomial time. Because the knapsack problem is an NP-complete problem, this implies that the constrained path problem is also NP-complete.