Algorithms 2
Exercise 3

1. Multiple Knapsack Problem

Consider the following version of the Knapsack problem, called the Multiple Knapsack Problem (MKP for short):

GIVEN: A set $S$ of $n$ items with value $v_i$ and weight $w_i$, and $k$ different knapsacks with capacities $B_1, \ldots, B_k$.

OUTPUT: Find a subset of $S$ with maximum profit (value) which can be packed into the $k$ knapsacks.

(Assume $P \neq NP$)

(a) Provide a pseudo-polynomial time algorithm which solves the MKP for $k = 2$ in $O(nB_1B_2)$ time.

answer

We construct a three dimensional table $A$ of size $O(nB_1B_2)$, such that $A[i, j_1, j_2]$ will maintain the maximum value of a subset of the first $i$ items, where the weight of the items in the first knapsack are no more than $j_1$, and the weight of the items in the second knapsack are no more than $j_2$. We fill $A$ in the following manner:

$$A[1, j_1, j_2] = \begin{cases} 0 & j_1 < w_1 \land j_2 < w_1 \\ v_1 & \text{otherwise} \end{cases}$$

and for $i > 1$:

$$A[i, j_1, j_2] = \max(A[i - 1, j_1, j_2], A[i - 1, j_1 - w_i, j_2] + v_i, A[i - 1, j_1, j_2 - w_i] + v_i)$$

The final solution will be in $A[n, B1, B2]$. We spend constant time per cell in $A$, which is of size $O(nB_1B_2)$. Note that if we only wanted the value, and not the subset, $O(B_1B_2)$ memory would suffice.

(b) Extend your algorithm for general $k$. Provide running time and space.

answer

We maintain a $k+1$ dimensional table $A$ of size $O(nB_1B_2\ldots B_k)$, such that $A[i, j_1, j_2, \ldots, j_k]$ will maintain the maximum value of a subset of the first $i$ items, where the weight of the items in the first knapsack are no more than $j_1$, and the weight of the items in the second knapsack are no more than $j_2$, and so on till the weight of the items in the $k$’th knapsack are no more than $j_k$. We fill $A$ in the following manner:

$$A[1, j_1, j_2, \ldots, j_k] = \begin{cases} 0 & j_1 < w_1 \land j_2 < w_1 \land \ldots \land j_k < w_1 \\ v_1 & \text{otherwise} \end{cases}$$
and for $i > 1$:

$$A[i, j_1, \ldots, j_k] = \max(A[i-1, j_1, \ldots, j_k], A[i-1, j_1-w_i, j_2, \ldots, j_k]+v_i, \ldots, A[i-1, j_1, \ldots, j_k-w_i]+v_i)$$

We spend $k$ time per cell in $A$, which is of size $O(nB_1B_2\ldots B_k)$ for a total of $O(knB_1B_2\ldots B_k)$. Note that if we only wanted the value, and not the subset, $O(B_1B_2\ldots B_k)$ memory would suffice.

(c) For the $MKP$ with $k = 2$, prove that there does not exist a pseudo-polynomial time algorithm which is polynomial in $n$ and $V = \sum_i v_i$ (HINT: recall that the partition problem defined below is NP-Hard).

**answer**

We will show that if such an algorithm exists, we can solve the partition problem in polynomial time. Given a multi-set $S = s_1, \ldots, s_n$ that we wish to partition into two multi-sets $S_1$ and $S_2$ such that $\sum_{s \in S_1} s = \sum_{s \in S_2} s$, we construct a set $S'$ of $n$ items, such that the value of the $i$'th item is 1, and the weight of the $i$'th item is $s_i$. We know run the algorithm that solves the $MKP$ with $k = 2$ on $S'$ where $B_1 = B_2 = \frac{\sum_{1 \leq i \leq n} s_i}{2}$. If the solution returned has value $n$, then we know that we can partition all of the numbers in $S$ as needed. If the value is $n - 1$ or less, we know that we cannot partition $S$ because, if we could, we would get a better value (the optimal would have been $n$).

If we have an algorithm that can solve the $MKP$ for $k = 2$ in time which is polynomial in $n$ and $V = \sum_i v_i$, then we would have a polynomial time algorithm which would solve the partition problem, because $V = \sum_i v_i = n$, so the algorithm would be polynomial in $n$, contradicting $P \neq NP$.

(d) Prove that the existence of a pseudo-polynomial time algorithm does not imply the existence of an FPTAS.

**answer**

We will show that an FPTAS cannot exist for this problem, unless $P = NP$. The idea is similar to the one from section (c). Noting that an FPTAS (by definition) will give us a $(1-\epsilon)OPT$ approximation for our problem, and will run in time $O((\frac{1}{\epsilon^2} n^\beta)$ where $\alpha$ and $\beta$ are two positive constants. If we set $\epsilon = \frac{1}{2n}$, the algorithm will run in time $O((2n)^{\alpha} n^{\beta}) = O(n^{\alpha+\beta})$ which is polynomial in $n$.

Now, given a multi-set $S = s_1, \ldots, s_n$ that we wish to partition into two multi-sets $S_1$ and $S_2$ such that $\sum_{s \in S_1} s = \sum_{s \in S_2} s$, we construct a set $S'$ of $n$ items, such that the value of the $i$'th item is 1, and the weight of the $i$'th item is $s_i$. We know run our FPTAS on $S'$ where $B_1 = B_2 = \frac{\sum_{1 \leq i \leq n} s_i}{2}$. If $S$ can be partitioned, the optimal value has value $n$, so our FPTAS will return an answer which is no less than $(1-\epsilon)n = (1-\frac{1}{2n})n = n - \frac{1}{2}$. If $S$ cannot be partitioned, the optimal values is at most $n - 1$, so the FPTAS will return an answer which is no more that $n - 1$, hence we can differentiate between the two in polynomial time, contradicting $P \neq NP$. 