Problem Set 3: Dynamic Programming Solutions

Due: Friday 10/29 11:59PM

Feel free to work in groups of at most 4 for these - if you have a group of more than 4, please run it by me first. If you do work in a group, please include the names of those that you worked with. **However: each student should submit a separate copy where the solutions have been written by yourself.**

1. Consider the following packing problem: items of varying weights $w_1, ..., w_n$ arrive over time (so the item of weight w_1 arrives before the item of weight w_2, and so on) and we must place them into boxes upon arrival. For each item, we can either place it into the current box we are filling if it fits, or we can close the current box and start a new box. Each box can hold at most W weight, and we cannot put an item into a box that is already closed. Define the slack of a box to be W minus the sum of the weights of the items in the box (so the amount of extra weight the box could still hold). Your goal is to give a polynomial time algorithm that packs boxes so that the sum of the **squares** of the slacks of all boxes (including the last one) is minimized. As an example, suppose $W = 5$, and $w_1 = w_2 = 2$ and $w_3 = w_4 = 3$. Then one possible solution might be packing the first two items in the first box and giving each of the last items their own box, giving slacks 1, 2, and 2 for a total cost of $1^2 + 2^2 + 2^2 = 9$. Another solution might be packing the first item in its own box, the next two into a second, and the last item in its own box, giving slacks 3, 0, and 2 for a total cost of $3^2 + 0^2 + 2^2 = 13$.

   (a) (5 points) Convince me that the algorithm from the first problem of the first problem set won't work for this problem, namely the algorithm that packs boxes as long as the weight will fit and starts a new box only when the next item won't fit in the current box.

   Solution: Consider the input $W = 5$, $w_1 = 3$, $w_2 = 2$, and $w_3 = 2$. Then the proposed algorithm would pack the first and second items into the first box, and start a new box with the 3rd item. This would give a cost of $0^2 + 3^2 = 9$. However, the optimal solution would be to pack the first item in its own box and the last two items in the second box, giving a total cost of $2^2 + 1^2 = 5$. Thus the algorithm does not match the optimal cost on this input, showing it is incorrect.

   (b) (10 points) Define an algorithm for this problem and convince me of its correctness.

   Solution: I saw a few good solutions for this problem, so I'm including a couple different ones:

   Let $\textsc{Opt}(i, k)$ for $i \leq n$ denote the optimal cost over items i to n, where the first box has k weight remaining to fill. Then if $w_i > k$, we must start a new box with the ith item, so $\textsc{Opt}(i, k) = k^2 + \textsc{Opt}(i + 1, W - w_i)$. Otherwise, if $w_i \leq k$, then we could instead put the ith item in the current box using up w_i of the k slack available. Thus $\textsc{Opt}(i, k) = \max(\textsc{Opt}(i + 1, k - w_i),\ k^2 + \textsc{Opt}(i + 1, W - w_i)$. We set $\textsc{Opt}(n + 1, k) = k^2$ as a base case: when there are no more items to pack, the remaining weight in the current box is the only slack.

In order to compute $\textsc{Opt}(i, k)$, we create an $(n+1) \times (W+1)$ array A. The following pseudocode computes A (we let $w[i] = w_i$ for notational purposes):

```
for k = 0...n-1:
    A[n+1, k] = k^2
for i = n...1:
    for k = 0...W:
        if(w_i > k):
            A[i, k] = k^2 + A[i+1, W-w[i]]
        else:
            A[i, k] = max(k^2 + A[i+1, W-w[i]], A[i+1, k-w[i]])
```

At the end of the day, the optimal solution's cost will be stored in $A[1, W]$. In order to get the solution itself, we need to backtrack from $A[1, W]$. Suppose we are at some current value $A[i, k]$, and we are trying to decide if item i started a new box or not. If it started a new box, then $A[i, k] = k^2 + A[i+1, W - w[i]]$, otherwise $A[i, k] = A[i+1, k - w[i]]$. Hence, we can backtrack as follows:

```
def backtrack(i, k):
    if A[i, k] = k^2 + A[i+1, W-w[i]]:
        start a new box with item i
        backtrack(i+1, W-w[i])
    else:
        put item i in the current box
        backtrack(i+1, k-w[i])
backtrack(1, W)
```

Another possible idea: Consider the solution tree where level j contains each possible partitioning of the first j items into boxes, and consider the following pruning rules:

i) If any partitioning contains a box where the sum of the weights inside that box is greater than W, we can prune.

ii) If two partitionings on the same depth have the same total weight in the last box, then we can prune the solution that has higher sum of squared slacks at that point.

The first follows from the constraint that each box can hold at much W weight, so once you've filled a box more than that no solution can follow. As for the second, consider two partial solutions S_1 and S_2 on level j where the last box of each solution has the same contents and let $s_1 \leq s_2$ be the sum of squared slacks for S_1 and S_2 respectively. Consider any solution $\tilde{S} = S_2 \cdot S_3$ extended from the partial solution S_2, where S_3 represents the choices on items after j within solution S:

then we could also consider the solution $S = S_1 \cdot S_3$, which is a feasible solution since the last box of S_2 and S_1 have the same contents. Moreover, since $s_1 \leq s_2$, the sum of slack squares in S is less than or equal to the sum of squared slacks in $\tilde{S}$. Hence we can prune the subtree of solutions rooted at S_2, since every solution in that subtree has a corresponding better solution in the subtree rooted at S_1.

As such, we will create an array A that builds solutions according to this tree. We will construct A as an $n \times W$ array, where $A[j,k]$ = the minimum sum of squared slacks over the first j items with k slack left in the last box. Note that if we have a solution on the first j items with k slack left in the last box, we have two options for the next item as per the solution tree: if it fits in the current box (if $w_{j+1} \leq k$), then we have can create a feasible option for $A[j+1, k-w_{j+1}$ by putting item $j+1$ into the same box. Alternatively, we could put item $j+1$ into a new box on its own, giving us a feasible option for $A[j+1, W-w_{j+1}]$. This induces the following pseudo codecode:

```
1 Initialize A as an n × W array with every entry set to ∞
2 Initialize A[1, W] = (W − w_1)^2.
3 for j = 1...n-1:
4     for k = 1...W:
5         if w_{j+1} ≤ k:
6             A[j+1, k-w_{j+1}] = min(A[j+1,k-w_{j+1}],
7                                     A[j, k] - k^2 + (k-w_{j+1})^2)
8         A[j+1, W − w_{j+1}] = min(A[j+1, W − w_{j+1}],
9                                     A[j, k] + (W − w_{j+1})^2)
```

At the end of the day, the min cost of any solution will be stored at the value $\min_{\texttt{k}}\texttt{A[}n\texttt{, k]}$. In order to get the solution itself, we back track from this point: note that if $W - \texttt{k} > w_j$, then w_j must be in a box with other items. The only other case is that it started a new box, in which case $\texttt{k} = w_j$ and there must be some value `newK` such that `A[j-1, newK] = A[j, k] - k`^{2}

```
1 def backTrack(j, k):
2     if W − k > w_j:
3         item j was packed in same box as previous
4         backTrack(j-1, k+w_j)
5     else:
6         item j started a new box
7         Find newK such that A[j-1, newK] = A[j, k] - k^2
8         backTrack(j-1, newK)
```

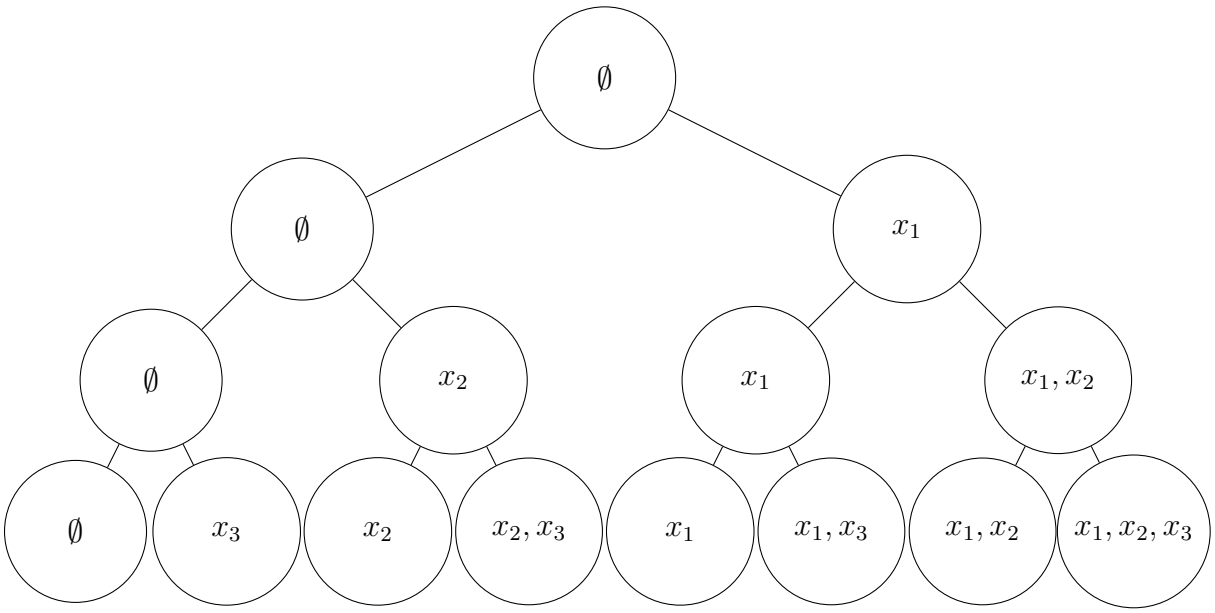
2. Assume that you are given a collection $B_1, ..., B_n$ of boxes. You don't know the individual weights, but you know that each box weighs some integer value between 1 and W pounds. You have also been given a pan balance: this device contains two platforms on which you can place as many boxes as you would like. After placing the boxes, it can tell you which side is heavier. Your goal is to determine if the boxes can be divided into two subcollections (with no overlap) such that the two subcollections have equal weight.

(a) (5 points) Consider the algorithm that tries every subcollection against every other disjoint subcollection using the pan balance. How many uses of the pan balance would this amount to?

Solution: What really matters here is that the number of uses is non-polynomial. In the very least, every subcollection would need to be tried at least once (actually much more than once for most, but just once gets us what we need): since there are 2^n subcollections, this is already way too many uses to give us a polynomial number of uses.

(b) (10 points) Determine a strategy to determine whether there are two disjoint subcollections of equal weight that uses the pan balance at most $O(n^2W)$ times and convince me of its correctness.

Solution: Consider the tree of all possible solutions:



Here, we are keeping track of every possible collection of the boxes. Note that since each box weighs a integer weight between 1 and W, and there are n boxes, the weight of all the boxes together is at most nW. Hence every subcollection weighs some integer between 1 and nW, so there are at most nW different possible total weights of sums of boxes. In accordance with our tree, we are going to use the pan balance to determine the sorted order of each level. Note that if we have sorted level j, we could observe whether there are any collections of the same weight from the first J numbers by just looking at consecutive collections in sorted order. As pointed out above, there could be at most nW different possible total weights of sums of collections of boxes, so as long as we haven't found a solution we know that there are at most nW possible collections per level.

Suppose we have sorted level j, and we want to know how to sort level $j+1$: notice that level $j+1$ is effectively level j along with every x_{j+1} appended to every element from level j. We know how to sort level j, and hence we'd know how to sort all the items from level j with x_{j+1} appended without any additional uses of the pan balance - now we just need to know how to merge them. From

our analysis of mergesort, we know we can merge these two lists with a number of pan-balances uses linear in the number of items in level j. Thus we can sort each level in $O(nW)$ uses of the pan balance.

Since there are n levels, this amounts to $O(n^2W)$ uses of the pan balance over the whole tree to keep track of the sorted order of each level. If we haven't found two equal collections after n levels, then there are no equal weight collections.