

CS375 Weekly Assignment 7

Due April 14th 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

You are encouraged to discuss the problems with your classmates, **but your submission should be your own**. When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others are from our course texts as listed on the syllabus.

1. In class we considered the maximal independent set problem in a path: in this problem, we're given values $v_1, \dots, v_n$ and must choose the subset of these values of maximal combined value such that we don't choose two consecutive indices. **First, argue that the following recurrence is correct:** let $\text{OPT}[i]$ denote the optimal combined value from the values $v_i, \dots, v_n$; then $\text{OPT}[i] = \max(v_i + \text{OPT}[i+2], \text{OPT}[i+1])$ for $i < n-1$ and $\text{OPT}[n] = \max(v_n, 0)$ and $\text{OPT}[n-1] = \max(v_{n-1}, \text{OPT}[n])$.

Solution: If the only items are $v_i, \dots, v_n$, then if the optimal solution includes v_i it also includes the optimal solution of the values $v_{i+2}, \dots, v_n$; if it doesn't include v_i , then it is simply the optimal solution of $v_{i+1}, \dots, v_n$. This gives the above recurrence.

Now that we have the recursive formula, we want to build an array that stores the values. One of the two algorithms works, the other does not. Which is which and why?

<pre>A = new array of length n + 1 A[n] = max(v_n, 0) A[n-1] = max(v_{n-1}, A[n]) for i in range(n-2, 0, -1): A[i] = max(v_i + A[i+2], A[i+1])</pre>	<pre>A = new array of length n + 1 A[n] = max(v_n, 0) A[n-1] = max(v_{n-1}, A[n]) for i in range(1, n - 1): A[i] = max(v_i + A[i+2], A[i+1])</pre>
---	---

Solution: The problem with the one on the right is that $A[i]$ is dependent on entries of A that *have yet to be calculated*; in Java, eg., these values are still 0 (assuming an int array). The one on the left works because we're calculating entries for A using our recursive value on entries we've already calculated.

2. **Buying Candy on a Budget:** As a teacher preparing for Easter on a limited budget, you want to maximize the total "candy joy score" of the candies you can afford. You have a budget of B dollars and access to n types of candy. Each candy type i has a price p_i , a joy score j_i , and unfortunately because it's (actually now past) Halloween there's only 1 bag left of each. Determine the combination of candies that maximizes the total joy score without exceeding your budget or the available supply of each candy type.

To set up the parameters more formally:

- Let B denote the total budget available.

- Let n denote the number of candy types.
- For each candy type i (where $i = 1, 2, \dots, n$):
 - p_i : the price of the candy of type i .
 - j_i : the joy score of the candy of type i .

Objective:

Choose a collection of the candies to maximize the summed joy score, subject to staying within your budget.

- (a) First, consider the following algorithm:

BestValue(p, j, B):

```

Sort the values in  $j$  so that  $j_1 \geq j_2 \geq \dots \geq j_n$ 
/** (note that we'll need to simultaneously
    * sort  $p$  to keep the corresponding values
    * in the same order) **/

output = empty list

for  $i$  in range  $(1, n + 1)$ :
    if  $p_i \leq B$ :
        add  $i$  to the output
         $B \leftarrow B - p_i$ 

return output
```

In plain English (as if describing this to a friend without any knowledge of CS), explain how this algorithm chooses which pieces to buy.

Solution: This algorithm repeatedly takes the remaining item of greatest joy that still fits in our bag.

- (b) Show that the algorithm from (a) is incorrect.

Solution: Let $B = 10$ and consider the following table:

Item	1	2	3
Joy	5	4	3
Price	10	1	1

The above algorithm will only take the first item, getting a total joy of 5, but we could have afforded items 2 and 3 to get a total joy of 7.

- (c) Design an algorithm of $\Theta(n2^n)$ runtime that correctly solves this problem (the correctness of this algorithm should be immediately obvious).

Solution:

```

subsets = GenerateAllSubsets(all the items)
bestSubset = None
for each subset in subsets:

    if priceOf(subset) ≤ B
       and (bestSubset == None or joyOf(subset) > joyOf(bestSubset)):

        bestSubset = subset
return bestSubset
```

This algorithm tries every possible subset and records the one of best value whose total cost is under the budget. To understand the runtime, the first step of generating all subsets takes $\Theta(n2^n)$ time. It then iterates over all possible subsets (of which there are 2^n) and for each one calculates

its cost, and the possibly sums its joy. Since each subset is of size at most n , we get a total runtime of $\Theta(n2^n)$.

- (d) Suppose every price p_i and B are all positive integers. Design a dynamic programming algorithm to solve this problem in a runtime that is polynomial in n and B (score will be based on how minimal this polynomial is along with space efficiency). Your answer should include an explanation of the algorithm's correctness and an analysis of the runtime of the algorithm. Include in your explanation where exactly you are using the assumption that all the p_i s and B are integers.

Solution: We will parameterize the problem by computing the following formula:

$\text{OPT}(i, b)$ = the best joy we can get from just the first i items with a budget of b .

We thus need to compute $\text{OPT}(n, B)$ and we will do so by deriving a recursive formula for OPT . Note that $\text{OPT}(i, b)$ either includes the i th item or doesn't:

- if it includes the i th item, then the best we can do is to get the best joy on the remaining $i - 1$ items with a budget of $b - p_i$
- if we don't include the i th item, then we still have a budget of b and just want the best joy we can get on the remaining $i - 1$ items.

Of course, if we don't have any items (if $i = 0$) then we can't get any value, and if the i th item's cost is greater than our budget (if $p_i > b$) then we can't possibly purchase it. This gives us the following recursive formula:

$$\text{OPT}(i, b) = \begin{cases} 0 & \text{if } i = 0 \\ \text{OPT}(i - 1, b) & \text{else if } p_i > b \\ \max(\text{OPT}(i - 1, b), \text{OPT}(i - 1, b - p_i) + j_i) & \text{else} \end{cases}$$

Now that we have a recursive formula, let's do some dynamic programming. We will first build a table **A** such that $A[i, b] = \text{OPT}(i, b)$:

```
A = new array of size (n + 1) × (B + 1)
set A[0, :] = 0
for i in range(1, n + 1):
    for b in range(B + 1):
        A[i, b] = A[i - 1, b]
        if p_i ≤ b:
            A[i, b] = max(A[i, b], A[i - 1, b - p_i] + j_i)
```

Note that our algorithm starts by handling the base case of $i = 0$, then uses the recursive formula for all remaining entries, relying on the fact that we are only looking entries on previous rows *which we have already computed*. Note that construction of **A** takes $\Theta(nB)$ time: for each entry of the table, we only need a constant time to compute it (because we're relying on precomputed entries).

Of course, this algorithm isn't complete - we've built **A** to tell us the optimal *value*, but we don't know the optimal solution yet. So we'll need to traceback:

```
// this proceeds having built A as above
i = n
b = B
output = []
while i > 0:
    if A[i, b] == A[i - 1, b]: // didn't take item
        i -= 1
    else: // take item
        add item i to output
        b -= p_i
        i -= 1
return output
```

This algorithm works on relying that we computed each entry according to our recursive formula - so we know that each entry came from one of two places. If $A[i, b] == A[i - 1, b]$, then we know that we didn't take the i th item; otherwise, we did take it. Because our while loop starts with $i = n$ and finishes when $i = 0$ and i decrements by 1 per iteration, our while thus has n iterations each of which takes constant time. Hence this traceback portion takes $\Theta(n)$ time, meaning the whole algorithm takes $\Theta(nB) + \Theta(n) = \Theta(nB)$ time.

- (e) Trace through your above algorithm on the input
- $B = 50$
 - $p = [16, 42, 5, 33, 7, 23, 6, 20, 50, 48]$
 - $j = [10, 19, 1, 18, 6, 17, 3, 14, 24, 23]$

I'm expecting a grid of values as built by your algorithm (you don't have to do this by hand, my solution just printed it by writing code for the algorithm you do above), along with a traceback computed (you could highlight the path through the array taken by your algorithm in the back-track step, maybe give an extra highlight for the steps that add something to an output...)

Solution:

[illegible]

Output is [8, 6, 5], meaning the 5th, 6th and 8th items.

- (f) Suppose instead of the prices and budget being positive integers, I promised all the j_i s were positive integers. Can you still come up with a dynamic programming algorithm? If so, what is the runtime now a function of? If not, why?

Solution: Since we have that the joys are the positive integers this time, it basically means we're stuck with using them as the 'indices' in our array. This is frustrating because our goal is to maximize the joy, but let's see what we can do.

Instead of trying to compute a function that determines the maximal joy we can get, let's instead try to compute something that tells me how much money I need to achieve a given amount of joy. Then, at the end of the day, we'll just find the maximal amount of joy that stays under our budget. For that purpose, let $J = \sum_{i=1}^n j_i$ be the sum of all joys. J is the maximal amount of joy that's even possible from buying all the candies, but it might not be under budget. As such, we'll try to compute the following:

$\text{OPT}(i, k)$ = the minimal budget necessary to get at least k total joy from the first i items.

Unfortunately, it's not as easy as just computing $\text{OPT}(n, J)$: this value might be more than our budget! In fact, we know exactly what $\text{OPT}(n, J)$ is: it's just $\sum_{i=1}^n p_i$; the only way we can get J total joy is to buy everything. But the input would be pretty boring if our budget were so big that $\sum_{i=1}^n p_i \leq B$: so we'll handle the more interesting case, where we'll need to find the largest value k such that $\text{OPT}(n, k) \leq B$.

But before that, we'll need a formula by which to compute OPT. Note that I can get 0 joy with a budget of 0 - I don't even need to buy anything! This is true for any non-positive value of k , in fact. However, if $i = 0$ and $k > 0$ (if I have no items but want to get any positive amount of joy) then no amount of funds will ever get me there; even infinite money. However, if $i > 0$ and $k > 0$, then to get a total of k joy I must either

- not buy the i th item and get a total of k joy from the remaining $i - 1$ items, or
- buy the i th item and get a total of $\max(k - j_i, 0)$ joy from the remaining $i - 1$ items. Note that I can take a maximum with 0 because getting at least a negative amount of joy requires the same budget as getting 0 joy.

Whichever option needs less of a budget is the option I want. This gives us the following formula:

$$\text{OPT}(i, k) = \begin{cases} 0 & \text{if } k = 0 \\ \infty & \text{else if } n = 0 \\ \min(\text{OPT}(i-1, k), \text{OPT}(i-1, \max(k-j_i, 0)) + p_i) & \text{otherwise.} \end{cases}$$

Let's calculate this formula using dynamic programming:

```
A = new array of size (n + 1) × (J + 1)
set A[0, :] = ∞ // case 2 above
set A[:, 0] = 0 // case 1 above (technically order mattered here)
for i in range(1, n + 1):
    for k in range(1, J + 1):
        A[i, k] = min(A[i - 1, k - 1], A[i - 1, max(k - j_i, 0)] + p_i)
```

We handle the base cases first, then use our recursive formula to evaluate each entry *which is just a function of entries on the previous row, which we've already computed*. Note that construction of this table takes $\Theta(nJ)$ time, since each entry takes constant time to compute.

We now just need to find the best joy I can purchase under my budget, then backtrack to find the items which give that joy:

```
// this proceeds having built A as above
i = n
k = J

while A[i, k] > B:
    k -= 1
// the above could be improved using binary search!
// but it won't matter in the grand scheme of things

output = []
while i > 0:
    if A[i, k] == A[i - 1, k]: // didn't take item
        i -= 1
    else: // take item
        add item i to output
        k -= j_i
        i -= 1
return output
```

The first while loops takes $O(J)$ time, and the second while loop takes $\Theta(n)$ time. This gives us a total of $\Theta(nJ) + O(J) + \Theta(n) = \Theta(nJ)$ time.