B01334462 () on time CS 157 – Fall 2017
B01160899 () **Oral Presentation** November 10, 2017

Problem 1: Knapsack

The knapsack problem essentially asks, given a weight limit W and n items $\{w_1, w_2, ..., w_n\}$, where each item has a value v_i, how do you maximize the value of the items you can carry?

Although it's slightly different from the book, let's look at this like a table T with the recurrence:

$$\text{T[i, w]} = \max \begin{cases} T[i, w - w_i] + v_i & (w_i \le w) \\ T[i-1, w] \end{cases}$$

So I can continually use the same item or I can choose other items. This'll create a $n \times W$ table, where each cell in the table $T[i, w]$ represents the max value of items that can be carried of weight w using only items $\le i$. This'll have run time $O(nW)$.

Likewise, let's update the above recurrence so that we can't use the same item more than once:

$$\text{T[i, w]} = \max \begin{cases} T[i-1, w - w_i] + v_i & (w_i \le w) \\ T[i-1, w] \\ T[i, w-1] \end{cases}$$

This'll have a similar run time as the algorithm above.

Run time What's tricky here is that this problem is actually **NP**-hard — it can be solved in *pseudo-polynomial* time, but since parameter W isn't a number of items, but rather a magnitude, it could result in exponential run time. If W is represented in binary, then increasing its length by one multiplies the possible number of W values by 2. The goal of the problem, then, is to rewrite it where we can define the complexity without W.

Rethinking Instead of thinking of the table as O(nW), let's instead create a table O(n(nV)) where V is the largest value for any weight. Our table, then, will look to minimize the weight needed to attain a certain value. If we iterate through the objects, we get:

$$\text{T[i, v]} = \min \begin{cases} T[i-1, v - i_{weight}] + i_{weight} \\ T[i-1, v] \end{cases}$$

The correct answer, then, is the largest value in the table that's less than W_{max}. Using this, our complexity is $O(n^2V)$.

Approximation Taking from Wikipedia, for some $\epsilon > 0$, let's define $K = \frac{\epsilon P}{n}$. This is going to give us some multiple value. If we then take this and for each object reevaluate its value:

$$v_i' = \text{floor}(\frac{v_i}{K})$$

we notice that, without applying floor() $K \times v_i' = v_i$. If K divides v_i, then the above is true. If it doesn't, $v_i' \times K$ is at most K less than v_i. So each of the n objects is at most K away from its actual value.

This means that, given any optimal strategy, OPT, and our approximate strategy, A, the value of OPT equals:

$$V(OPT) - K \times V(A) \leq nK.$$

So, the actual solution from our dynamic programming, S', will be in between V(OPT) and have a value at least greater than $K \times V(A)$:

$$V(S') \geq K \times V(A)$$

We can then rewrite this:

$$\begin{gathered} V(S') \geq V(OPT) - nK \\ V(S') \geq OPT - \epsilon P \\ V(S') \geq OPT - \epsilon OPT \\ V(S') \geq (1 - \epsilon)OPT \end{gathered}$$

We can do the third step, since $OPT \geq P$ — you can't have a P with a weight that wouldn't, by itself, fit in the knapsack. OPT, them, will be at least P plus possibly some other values.

We had a running time of $O(n^2 P)$. Every P we divided by K , where $K = \frac{\epsilon P}{n}$. Plugging in, we get:

$$O(n^2 \frac{P}{K}) = O(n^2 \frac{n}{\epsilon}) = O(\frac{n^3}{\epsilon})$$

which is a running time that no longer depends on a magnitude, P, and is therefore fully polynomial.

Problem 2: Parts

Algorithm: I think the key here is to sort the parts by:

1. The position of the first '+' in the part.
2. Where, if there's a tie we check succeeding symbols and sort based on precedence: $\emptyset \rightarrow + \rightarrow -$

Intuition behind this is that any other sorting might lead to an invalid ride. If we simply sort by the overall height of the part we might try to start the ride with: -++++++, which wouldn't work. The only sorting method that'll ensure we don't dip below ground takes into account the position of the '+'s.

Given a list of parts, let's sort according to the above criteria. This'll give us a list of parts, L, sorted in order of increasing position of '+'s. Once we have L sorted, let's work with the following recurrence:

$$\text{T[i, h]} = \max \begin{cases} T[i-1, h - i_{height}] + i_{size} & \text{if } T[i-1, h - i_{height}] + i_{height} = h \\ T[i-1, h] & \\ i_{size} & \text{if } i_{height} = h \end{cases}$$

We'll create a table, T, with $n+1$ rows corresponding to the parts and $h+1$ columns, where h is the max height possible from all n parts:

$$h = np_{max} \mid p_{max} \text{ is the longest part}$$

Each cell of T will represent the maximum number of parts used to attain a height of h. Row 0 and column 0 will be pre-populated with zeros, while the rest of the table will be filled with $-\infty$. We'll fill our table from top to bottom, right to left, with the maximum number of parts needed to create a viable ride ending up in table position $T[n, 0]$.

Example: The example from the question provided parts:

$$[- - + + +, + +, +, - - -] \rightarrow [+, + +, - - + + +, - - -]$$

Our initial table will therefore look like:

	0	1	2	3	4	...	20
	0	0	0	0	0	...	0
$+$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	...	$-\infty$
$+ +$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	...	$-\infty$
$- - + + +$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	...	$-\infty$
$- - -$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	...	$-\infty$

As we fill, it'll look like this:

	0	1	2	3	4	...	20
	0	0	0	0	0	...	0
$+$	0	1	$-\infty$	$-\infty$	$-\infty$	...	$-\infty$
$+ +$	0	1	2	3	$-\infty$	...	$-\infty$
$- - + + +$	0	5	6	7	8	...	$-\infty$
$- - -$	10	11	6	7	8	...	$-\infty$

Proof Let's prove that the above algorithm works via induction in (i, h) traversed in the same way as the algorithm states. Note the base cases:

1. when we have 0 parts, our ride can't ever get off the ground, so we'll fill in row 0 with zeros

2. before we start, we know the longest valid ride is of length 0, since this starts on the ground, ends on the ground, and never goes below the ground.

Consider an arbitrary list of parts, L, that have been sorted as above. Now consider an optimal solution that has correctly filled in cell (i, h). Consider the last decision this solution made. When filling (i, h):

1. part i could have the same height as h, in which case we'll set T[i, h] $= i_{size}$

2. part i plus some other maximal number of parts will have the same height as h, in which case T[i, h] $= T[i-1, h-i_{height}] + i_{size}$

3. we can't have a height h that includes part i, in which case we'll use the previous maximum number of parts not including i

Since we're taking the max of all the possible ways to construct a ride of height h, we know that (i, h) will be filled with the maximum number of parts, using $0 < k \leq i$ parts, to reach height h.

As a note, since T doesn't include negative heights, we'll never even consider rides that dip below the ground, thus ensuring all rides are valid.

Problem 3: TS

In this problem, we're trying to figure out a way to place T. Swift and her friends (and enemies) into trailers — all n of them. Let's try to find a polynomial time algorithm to do this.

Algorithm Let's take the hint from the problem that we should try to find the *left-most* person. Regardless how we order the people, someone **has to be** on the end. Importantly, this person can have at most 10 friends — most others can have 20: 10 people to the left and 10 to the right of them.

1. For all n, let's pick someone, p_1, who likes ≤ 10 others where everyone in S_1, where S_i is the set of people p_i likes, also likes each other. We can do this in constant time per person since $d = 10$, so $\mathbf{O(n)}$.
2. Next, for all $p_i \in S_1$, sort them by size of their remaining sets where $S_i = S_i - ((S_1 - p_i) \cup p_1))$. You do this since every one 10 trailers from p_1 must also like each other, so we can remove these people from the remaining set of people p_i needs to be placed close too.
   (a) If any $|S_i| > 10$, we won't be able to place them next to all their friends (it could mean they initially liked > 20 people) pick a different n from step (1).
3. Place p_i with larger $|S_i|$ farther to the right (in reference to the 10 available slots to the right of p_1). The rationale behind this: if after being placed into their trailers, $|S_j|$ and $|S_k|$ equal 7 and 8, respectively, the 8^{th} person p_k likes can't be within 10 trailers of p_j or p_1. If p_k is farther to the right, then position of $p_k + 10 > p_j + 10$.

   If, instead, we placed p_j farther to the right, $p_k + 10 \leq p_j + 10$, meaning the 8^{th} person p_k likes will be within p_j's range, which is invalid.

   Specifically, let's look at the size of the remaining sets and place them in trailers in increasing order, left $\rightarrow$ right.

   Note: If two people only like each other, add 10 empty trailers to the side to separate them from the others. If one person's set is 0 while the other's isn't, not everyone in the set likes each other, so try a different n.

   Going from left to right, there should be $1, 2, 3, ..., 10$ elements remaining in each of the sets, which we'll now rename $S_1, S_2, ..., S_{10}$. First, place the element from S_1 ten spaces to the right. When you do, check to make sure the element is in all other sets

$S_2, S_3, ..., S_{10}$, removing it each time. If it isn't in any set, choose a different n. Continue this process until everyone has been placed in a trailer.

Proof Let's prove this via induction on the number of people n. In the base case, where you have 0 or 1 person, placing them in the trailers is trivial. Let's assume for some $0 \leq i \leq n$, we can correctly place all i people into the trailers. Let's show that with the above algorithm, we can also determine a correct solution for $i + 1$ people.

Run Time