Francisco Rocha
CMPS 102
2/10/15

Homework #2 (Greed)

1. Given a sequence of cars S that will drive by, and the subsequence P that Betty is looking for that day, give an algorithm that detects whether Betty will be happy or sad in $O(|S| + |P|)$ time.

The algorithm we will use to determine whether Betty will be happy or sad at the end of the day is as follows:

> Start at the beginning of Betty's list $P = \{p_1, p_2, \dots, p_n\}$ and as cars go by in sequence $S = \{s_1, s_2, \dots, s_m\}$,cross out item p_i if it is the color she is looking for and begin looking to see if next item on Betty's list, p_{i+1} goes by.
>
> If you run off of Betty's list *P*, she will be happy.
> If you run off of sequence *S*, she will be sad.

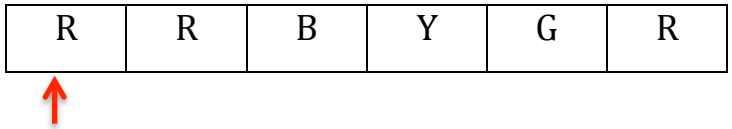
Proof of Correctness and Runtime:

The invariant to notice in the algorithm above is that we will never be looking for an item p_i in until everything previous to p_i has been seen in S. In other words while we will always advance the pointer in S we **only** advance the pointer in P to p_{i+1} $(for\ 1 \leq i \leq n, where\ n\ is\ number\ of\ elements\ in\ P)$ if during any iteration $p_i == s_j (for\ 1 \leq j \leq m, where\ m\ is\ number\ of\ elements\ in\ S)$. **This does not to indicate a nested for loop** (think of a while loop with two variables used as pointers and a check to see if P or S has "run off" where we define "run off" to occur when a pointer exceeds the last element in list.) Since you are traversing both lists (or arrays, if you will) once, therefore the runtime will be $O(|S| + |P|)$ and we will know whether Betty is happy or sad depending on whether S or P resulted in a run off. We don't really care for anything in S that doesn't match P since we are looking to see is P is a subsequence of S where we define a subsequence to be a desired order of appearing elements that are not necessarily contiguous. Therefore if up to a

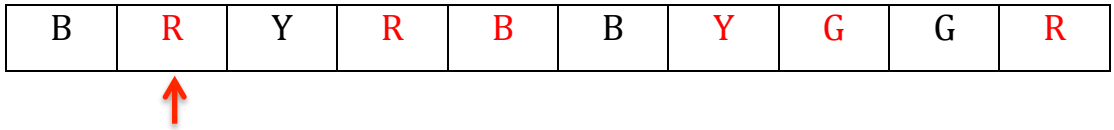
certain point s_j we have found everything in P from $p_1 to\ p_i$, then the next matching, if found, will depend on whether said item p_{i+1} is identical in s_j and so forth. Thus by the definition of a subsequence, our invariant will hold at every iteration in P.

Pictorial Example:

P:



S:



2. The farm is on the move and Farmer Mike has a truck that can hold up to W pounds without bottoming out and each cow has an individual weight w_i. Given that for every $i < j$.

The algorithm that determines the least number of trips Farmer Mike has to take in order to transfer all of his cows to Saskatchewan is as follows:

Ship cows to new farm based on pecking order and without overloading truck with weight W, during trip t.

ie.

- Let cows be sorted by pecking order with weight w.
- For w_i, check if w_{i+1} can fit on trip x without exceeding W
  - Yes?
    - check if next cow ($ex.\ w_{i+2}$) can join them in trip t
  - No?
    - leave to new farm with given cows in truck.

Proof of Correctness and Runtime:

We will prove that our algorithm, A, is correct by proving that it "stays ahead" from any other optimal solution O. Since our algorithm A packs as much into any given trip t, let O have cows $c_1, c_2, \dots, c_i$ in trip t and A have cows $c_1, c_2, \dots, c_j$ where $i \leq j$. We shall prove that our algorithm A is actually optimal by induction on the number of trips it takes, t.

Base Case:

It only takes one trip, $t = 1$, then our algorithm A will fit as many cows as it can into said trip.

Induction step:

For all trips $t > 1$, let us use our assumption that O fits $c_1, c_2, \dots, c_{i'}$ into the first $t - 1$ trips while A fits cows $c_1, c_2, \dots, c_{j'}$ where $i' \leq j'$. On the t^{th} trip we can observe that O will be able to fit cows $c_{i'+1}, \dots, c_i$ while our algorithm A will be able to fit cows $c_{j'+1}, \dots, c_i$ and potentially more since $i' \leq j'$ and it might still have a certain amount of weight limit to spare while O will be maxed out at said point due to it deciding to not max itself out during one of the pervious trips. Proving this induction step though shows that our algorithm A stays ahead of O and is actually optimal since from this point forth O will be "catching up" while A will be "staying ahead".

Additionally we can see that the algorithm will run in $O(n)$ (linear time) since we are going though the ordered list of cows and simply checking to see whether adding the next cow on the list will exceed the weight limit W and at most we will do $|C|$ number of checks since we are only transferring each cow over to the new farm one time and simply checking to see which trip they are going to be able to partake in.

3. Order the cows milking time in order to finish all of the morning rituals as soon as possible. Mathematically, each cow has a list of n triplets (m_i, e_i, d_i) corresponding to milking time, eating time, and drinking time of the i-th cow.

The ordering which minimizes the time it takes to complete morning rituals is described my the following algorithm:

Milk the cows in order of decreasing $e_i + d_i$ to minimize morning ritual completion time.

Proof of Correctness and Runtime:

To prove that our algorithm A is optimal let us assume that some other solution O is another optimal solution to problem. We must show that if we change O to look more like A, lets call that change O', O' will be better or as good as O.

We will therefore assume that there is an adjacent inversion in O's ordering such that making such inversion will make O look more like A. Picture below clarifies:

O:

O':

Let's define O to have ordering where cow i gets milked before cow j $\left(i \xrightarrow{then} j\right)$, but $e_i + d_i < e_j + d_j$. Let us now make such adjacent inversion to be between cows i and j to make O look more like A. When we make the swap $(j \xrightarrow{then} i)$, cow j gets milked before cow i. In this ordering (O'), j will finish getting milked sooner than she used to. Additionally in O', i will finish getting milked at the same time j finished getting milked in O but given $e_i + d_i \leq e_j + d_j$, in O' i will finish its morning routine no later than j did in O therefore O' does not have a greater completion time than O. If we continue making such inversions we will eventually end up transforming O

into A, whose overall completion time is no greater that O, thus our algorithm is also optimal.

Also note that O' **is** a legal transformation of O since they differ by at least 1 **adjacent** inversion, therefor they both contain same number of elements and by commutativity $i + j = j + i$. Furthermore since i and j are accepted by O is an accepted solution making i and j legal elements in O', therefore inversion will yield a legal solution.

Additionally we can see that our algorithm will run in $O(n \log n)$ since the most costly operation we are performing is the sorting by decreasing order of $e_i + d_i$ and the best runtime for anytime of sorting is $O(n \log n)$.

4. Farmer Mike wants to deal with as small of a representative committee for the cow as possible while making sure that every cow is satisfied with such committee to help ease cow unrest. Each cow will be happy if at least one of member whose ideals interval intersects with its own. Let us help him find the smallest committee possible that satisfies entire cow heard.

The algorithm that picks out the smallest committee of cows while maintaining cow happiness is as follows:

Steps:

1st: Sort cow subintervals by increasing right most number, lets call that their "finish times".

2nd: In said ordering assign first cows "finish time" to be f.

3rd: From all following jobs who's "start time" s $(left\ most\ number)$ is less than or equal to f pick job j with the **maximum** finish time. If none exits then it chooses cow whose finishing time is f to be j.

4th: Move j to committee and remove **all** cows whose subinterval intersect with j from group of candidates.

5th: Repeat steps 1-4 until there is no more cows to choose from in group $(ie. they\ are\ all\ satisfied\ with\ the\ comittee)$.

Proof of Correctness and Runtime:

We will prove that our algorithm is correct and valid proving that is another optimal solution O that is said to be better that our algorithm A. Our claim is that we can make O look more like A while either improving it or having it remain just as good. Let us call changed optimal solution to O'. We must now prove that no only is O' a legal but that it is also as good or better than O after making it look more like A. For clarification, this means that up to a certain point A and O' will have chosen identical elements, picture bellow clarifies:

A:	a	b	c	x	q	r
O:	a	b	c	y	s	t
O':	a	b	c	x	s	t

O' is a legal solution:

O' is a legal solution because in our algorithm A element a certain element x was chosen by A while O at this point decided to take a certain element y. Since A chose x, and A's specifications allow it to only choose legal elements and since O and A are identical in their choosing up to this point, x was a legal choice that could have been made.

O' is as good or better than O:

O' is a solution that is as good or better that O since it contains element x rather than y and x is a better choice than y since it was chosen by our algorithm A who looks at the largest interval (earliest start time and latest finish time) that includes the first element in sorted list of cow subintervals. That means that y did have a start time less that job $j's$ finish time f and was considered by A, but A choose x because x had a later finish time than y. Therefore, O' is both a legal solution that is as good or better than O.

Additionally we observe the runtime is going to be sorting which will have a runtime of $O(n \log n)$ and just like previous question. All over operations will run at a faster runtime that this since you will either be iterating the list searching for start time $s < f$ followed by then searching for which one of located subintervals who meet previous criteria have the latest finish time and each search will take at most $O(n)$ thanks to our original sort. The removing of jobs will also take $O(n)$ running time if for example you have a cow who is very open to all ideals and whose subinterval covers everything.