**PROBLEM 1**

**ALGORITHM ANALYSIS**

* Let *S* be an array of n colors.
* Let *P* be an array which holds the colors which make Betty happy.

For farmer Mike, the algorithm needs to be able to check the arrays by comparing each color with every array. To be able to do this, each color must be checked from an array *S* so that it matches the color from an array *P*. If the arrays contain a match, then the pointers of both arrays must be moved so that it points to the next components of each array, however, if the beginning item of array *S* does not match the beginning of array *P*, then the pointer of array *S* should only be moved. From here, we will move onto checking if the following item on array *S* is a match to the item in array *P*.

* Begin with Betty’s list *P*, with an array *P* = (P1, P2,…,Pn).
* Every car that pass by in a sequence *m*, each will be referred to as (S1, S2,…,Sm).

With this, we can move on to each *Pi* if the color of the car is what makes Betty happy and then observe if the following car, *Pi+1*, also makes her happy.

There are two possibilities for Betty:

1. If Mike goes with Betty’s list, then she will be happy
2. If Mike goes with sequence *S*, then she will be sad.

Proof:

* If we assume that the algorithm above is valid and we don’t take into consider a car *Pi* until every single car that preceded *Pi* has been seen in the sequence *S*. Here, we can state that we will always increase the pointer in *S* when the pointer in *P* becomes *Pi+1*. This means that for every *i* in between 1 and *n*, *n* would indicate if the number of elements in the Bettys list *P*.
* Since two lists are being compare one at a time, the run time therefore is O(|*S*|+|*P*|).
* To check if Betty is happy or sad will be told by the elements of each list and whether these elements had been run off.
* The items that do not match the list of Betty will *not* be taken into account since we are trying to figure out if *P* is a sub-sequence of *S*
  + Where sub-sequence is the desired list of elements that is what makes Betty happy.

Runtime:

* Runtime = O(*n*), where *n* = number of elements in an array *S*.
* Worst case scenario = if the array pointer for the items in the array *S* goes to the end of the array and the pointer for the array *P* also goes to the end.
* Best case scenario = if the pointer of array *P* arrives before the pointer of an array *S*, since both of the pointers don’t have to match each color by the time the pointer of array *P* arrives at the end of the array.

**PROBLEM 2**

**ALGORITHM ANALYSIS**

* Let *n* be the number of trips that Farmer Mike has to take for his n cows.
* Let *C* be the strict pecking order *C1, C2, ..., Cn* and insist on traveling to the new farm in that order.

The truck that the farmer drives can hold up to W pounds without bottoming out and each cow has individual weight *Wi*. We need to use an algorithm that can minimize the number of trips that the farmer needs to have without bottoming out and the farmer should keep the pecking order based on individual weight of cow. Given that for every *i< j.*

The algorithm that we are going to use is to determine the least number of trips that the farmer has to take to transfer all of his cows to the new farm.

* Let *W* be an individual weight of each cow.

The farmer cannot overload truck more than the maximum weight W, and consider *t* represents the number of trips that the farmer needs to take.

* Let cows be sorted by pecking order so that the farmer can know how much weight *W* does each cow have.

For every *W*i, the farmer needs to check if *Wi+1* can be loaded into a truck without exceeding the maximum weight that truck can accommodate. If yes, then the farmer needs to check if the next cow can be fit into the trip, or if not, then the farmer needs to leave to transfer the given cows in a truck to a new farm.

Proof:

* We are going to prove that the algorithm above is correct by illustrating that the farmer has the optimal solution. Because we are trying to pack as many cows as possible into any trip t, let *O* have cows in *C1, C2,, Ci* in trip *t* and *E* have cows *C1, C2,,Cj* where *i* is less than *j.* We are going to prove that the algorithm that we have given works optimally by using induction on the number of trips it takes, *t*.
* Given a list of n things *Si*, and a list of c other things, design an algorithm that decides whether or not it is possible to do things, and nothing has more than |*n*| things. Assume that you can find “too far” or not.

*c*

Base Case:

* Assume the farmer takes one trip, *t=1*, and the farmer can load as many cows as he can fit into a truck and transfers to a new farm.

Induction Step:

* For all trips that is more than once, assume that algorithm *O* fits *C1, C2,..., Ci* into first t-1 trips for cows *C1, C2,..., Cj* where *i* is less than *j*. We are going to observe if *S* will fit cows (*Ci+1,...,Ci)* for each *tth* trip. For each *tth*trip, we can find that *O* will be able to accommodate cows (*Ci+1,...,Ci)* while the algorithm that we have given might be able to fit cows (*Ci+1,...,Ci)* and given that *i* is less than *j*, and possibly the farmer can fit more cows in one trip. There will be certain amount of weight limit to consider while the algorithm *O* will be maxed out during one of its previous trips. We have proved through the induction step that our algorithm is more optimal than algorithm *O* because *O* is catching up the algorithm we have come up with and also our algorithm is staying ahead.

Runtime:

* We know that the individual cow has a different weight and we need to use the algorithm above to sort out the weights of cows in the strict pecking order. In this problem, the truck can hold up to *W<4*, and we need to consider that we want the farmer to take a minimum number of trips.
* We know that the algorithm runs in *O(n)* because we are going through the ordered list of cows and check to see whether the next cow on the list can exceed the limited weight *W* that the truck can hold. We are going to do a number of checks because we are checking how many cows can fit in and transferring each cow over to the new farm one at the time.

**PROBLEM 3**

**ALGORITHM ANALYSIS**

* Let the algorithm *M* be the algorithm that orders the cows milking time so that all of the morning routines can be completed as early as possible. By using mathematical list, each cow has a list of triplets that states (*mi, ei, & di*), which represents milking time, eating time, and drinking time of the *ith* cow.

Algorithm:

* To minimize the morning ritual completion, the farmer should milk the cows in order of decreasing *ei+di*.

Proof:

* Prove that the algorithm is optimal.
* Solution *A*: possible alternative optimal solution to problem.
* We need to start off by showing that we can transform *A* to be something similar to the algorithm being used. *A* is another version of possible optimal solution to problem. *A* should be defined as the instance when cow *i* gets milked before another cow *j*, under the condition that *ei + di < ej + d*j. This adjacent inversion is required in between cows *i* and *j* so that we can make *A* to be closer to the algorithm. Swaping *j* to *i,* then we know that cow *j* gets milked first and then cow *i*.
* In algorithm *A*, consider the case that cow *i* finishes the milking procedure and cow *j* finishes getting milked simultaneously in *A*, but with the condition that *i* and *j* are not equals (*ei+di≠ej+dj*) in *A*. Here, cow *i* should finish its morning rituals faster than *j* did in the algorithm *A*. We can then conclude that *A* is not optimal to *A*. Continuing with this inversion procedure, eventually we will end up have *A* in our algorithm, which is not greater than *A*. Here, we can conclude that the algorithm is optimal.

Runtime:

* NOTE: *A* is a transformation of *A* by at least one adjacent inversion, so they both contain the same number of elements.
* Here, we know that the algorithm is *A*(*nlogn*) because what is being performed is the sorting by lowering the order of *ei+di* and the best case of the runtime then is *A*(*nlogn*).

4 Problem 4 - The Funny

4.1 Algorithm Analysis

Proof:

Runtime: