Faith Twardzik Orpaz Goldstein, 1F, 2-3:50 pm

1) Exercise 20 on page 329

We will use dynamic programming in order to give an algorithm which decides how many hours should be spent on each project so that the average grade received on those projects is maximized. We will store both the maximum grade possible at each level of the recursion, as well as how many hours should be allocated to the courses in question. So, we end up with the OPT solution:

OPT[course, hours] = [the maximum total grade possible for course *c* after spending *h* hours on said course] + OPT[*c – 1, hours – h*]

The above optimal solution will find the complete maximum grade for all of the courses, given that the average will be a fixed division by the total number of courses. So, the optimal solution will generate the maximum total grade for course *c,* and generate the specific number of hours (*h*) that should be dedicated in order to achieve this maximum, plus the optimal solution (the maximum total grade) for the previous course, with the hours equal to the total hours minus the hours dedicated to the current course. In order to satisfy the dynamic programming component, we will store both *h* (the hours spent on a certain course) as well as the value OPT[course, hours], so that when end our recursion with a value of OPT[*n, H*], where *n* is the total courses and *H* is the total hours, we can go back through our stored values and distribute the hours. We should also note that the base cases are such: OPT[0, *h*] = 0 (since a max grade of 0 can be achieved if there are 0 projects), and OPT[*courses*, 0] = the summation of the grades for each course with 0 inputted to the function as the hours parameter, which will obviously also equal 0.

Time Complexity: Since there will be *nH* total values stored, given that all of the courses must be considered, as well as all of the variations in hours dedicated to each course, and given that each level of the OPT solution will have a runtime of O(*H*), the total time complexity will be O(nH^2).

2) Exercise 21 on page 330

We will again use dynamic programming in order to provide k-shot strategies which maximize return given a sequence of prices. This will be a two-step process, as we are not only determining a way in which to generate the maximized profit from one buy/sell process, but also how to combine the buy/sell processes into one optimal strategy given the *n* days provided. Therefore, we begin by tackling the first issue (solving for a maximized profit based on one buy/sell interaction). In order to do this, we have the following recurrence:

OPTR[i, j] = the maximum of [R[i, j], OPTR[i + 1, j], OPTR[i, j – 1]]

In the above recurrence, we have OPTR denoting the maximum return/profit for one buy/sell interaction, where i is the day on which the stock is bought, and j is the day on which the stock is sold. This optimal return will be equal to the maximum between three values: the buy/sell return on days i and j, respectively, or the optimal return/profit from a buy on day i + 1 and a sell on day j, or the optimal return/profit from a buy on day i and a sell on day j – 1. So, based on all values for i and j, we will store all of the values for OPTR.

The second part of the question that we must tackle is in developing the best k-shot strategy. So, we must not only find the optimal return of *one* buy/sell interaction, but the optimal return of a *set* of

buy/sell interactions over the course of *n* days, where these buy/sell interactions do not overlap. So, we have somewhat of question of intervals to schedule. Therefore, we will use a recurrence set-up similar to that in the previous question, where:

OPTK[t, d] = the maximum of OPTR[i, j] + OPTK[t – 1, i – 1]

We have two base cases: one in which t = 0 and one in which d = 0, where the first would give an OPTK value of negative infinity, given that if there are 0 transactions, no profit is possible regardless of the days given and the second would give an OPTK value of negative infinity because if there are 0 days, 0 transactions are possible. In the above recurrence, OPTK denotes the optimal k-shot strategy, where t is the buy/sell transactions and d is the number of days (*n* in the given problem). We will look up/take the maximum of the current transaction, which we stored when we used the formula OPTR[i, j], and add it to the optimal solution OPTK for the previous transaction (t – 1) and the previous day (i – 1). We will store these values for OPTK, and the ending, optimal k-shot strategy will be equal to the stored value for OPTK[t, n].

Time Complexity:

The computations for generating and storing the OPTR values will have a runtime of $O(n^2)$, because we must calculate every set of intervals of days (with both a starting and ending day) that will result in the maximum return, plus the constant time maximum operation. Since we will be running through O(n) maximum returns and spending O(n) time on each to create the k-shot strategy associated with it, we end up with a total runtime of $O(n^2)$.

3) Exercise 24 on page 331

Here we are using dynamic programming in order to solve the problem of determining whether or not it is possible for gerrymandering to occur (for the creation of two districts, based on the divvying of precincts into each, where each has n/2 of the available precincts so that one party has a majority in both of the created districts). We will do this by storing a number of values. If we take *P* to be the party which may potentially have the majority of votes (if the precincts are susceptible to gerrymandering), then at each subproblem, we will store the votes for *P* currently in district 1, the votes for *P* currently in district 2, and the precincts both in district 1 and district 2.

We will do this in a Boolean format, where *P*[v1, v2, p1, p2, p] will be equal to true if the precincts cause potential gerrymandering, or false if it is not possible with the current scheme of precincts assigned to each of the districts. Then, we will need only to look up true values in our stored table to see which configuration/s of precincts in each district result in a gerrymandered result. Note: in *P,* v1 and v2 stand for the current votes in district 1 and 2, respectively, while p1 and p2 stand for the current precincts in districts 1 and 2, respectively. p stands for the number of precincts we have gone through so far.

So, for each subproblem, we will traverse through the precincts and assign one to either district 1 or district 2 and store the values for *P*. So, for the value of *P* at the next precinct to assign (*P*[v1, v2, p1, p2, p + 1], we will place p + 1 in either district 1 or district 2. Say that there are x votes for party *P* in the precinct p + 1. If we were to place it in district 1, we would need to check *P*[v1 – x, v2, p1 – 1, p2, p], or in the case that we send it to district 2, we would check *P*[v1, v2 – x, p1, p2 – 1, p]. After we have finished storing these values, we find a complete solution by searching through the entries in order to find a Boolean value of true. Thus, it will be the case that gerrymandering is possible if there is an entry that is

true, the total number of precincts p = *n*, and half of the precincts are in each district (p1 = p2 = *n/2*). A value of true for this *P* set-up indicates a configuration where gerrymandering has occurred.

Time Complexity: If we have *n* precincts and *m* voters, and for each of the precincts we are determining whether or not there is gerrymandering based on placing it in either district 1 or district 2, and since we are doing this so that each voter could be in either district 1 or 2 in each of the subproblems, we end up with a gigantic time complexity of $O(n^2m^2)$.

4) Exercise 8 on page 418

a) This will be an example of utilizing the Pre-flow-Push algorithm from page 360 of the textbook. However, before we utilize this algorithm, we must build a graph on which to use the algorithm.

- We can do this by first setting up the Source and Sink nodes, as in the labeling conditions from page 358. In this instance, the Source will be a node from which we will have four edges – from Source *S* to each of the supply types, *sa, sb, sab,* and *so.*
    - The capacities of the edges will be equal to the supplies given for each of the blood types, respectively.
- On the other end of the graph, the Sink node *Si* will have four edges – from Sink *Si* to each of the demand nodes, *da, db, dab,* and *do.*
    - The capacity for these edges will then be equal to the demands for each of these blood types.
- To set up the middle portion of our graph, we construct an edge from a supply node to a demand node if the demand node is able to receive blood from the blood type corresponding to the supply node.
    - The capacity for these edges will then be demand for the blood type corresponding to the demand node (if there is demand).
- Next, utilize the Preflow-Push algorithm in order to determine the maximum flow of the given graph.
- But, in order to see if there actually is enough blood and to evaluate if the blood will suffice for the projected need, we must use the results of the algorithm – we say that if the flow runs from the Source *S* node to the supply nodes, to the demand nodes, and reaches the Sink *Si* node (is saturated at this point), then we know that the capacities of the demands, given the capacities of the supplies, have been met. So, if all of the edges running from the demand nodes to the *Si* node are saturated, then we know that the maxflow has met our projected need, as the capacity of the supply would have run to the demand, and if it is enough, it will have run through the capacity of the edge connecting the demand nodes to the sink. If it was not enough, the flow therefore would not have reached the sink node.

Time Complexity: Because we are using the Pre-flow-Push algorithm, the total time complexity must be O(mn), if we use the most efficient version from the textbook (or $O(n^2m)$ if we use the generic one).

b) No, the 105 units of blood on hand is not enough to satisfy the demand, because the supply vs. demand needs do not match. However, we attempt to find a maximum possible allocation of blood to save as many patients as possible, which is as follows:

- Since blood type AB can only be given to AB patients, give 3 units to the 3 AB patients

- Since blood type B can only be given to AB and B patients (but we have satisfied AB patients), give 8 units to the 8 B patients
- Since O patients must take O blood, give 45 units to the 45 O patients. This leaves 5 units of O left.
- We now have 42 A patients in need of blood. Give the 36 units of A blood to 36 patients. Then, give the leftover 5 units of O to A patients. This satisfies 41 of the A patients, leaving only 1 A patient without any blood.

5) Exercise 12 on page 420

Here we are attempting to find a set of edges whose size is equal to k, where the maximum flow of the graph G' = (V, E – F) is as small as possible. So, we are looking at a graph G' which has F edges deleted from it, so that the size of F = k, and so that the maximum flow is minimized.

- So, we are given a graph with maximum s-t flow. Then, we will take two subgraphs that form a cut, where subgraph G1 has k edges deleted from it to form G1, and the other subgraph G2 (which is connected to subgraph G1) has the other edges.
- Then, if we had a maximum flow of *maxflow* in graph G, we will have a maximum flow of *maxflow* – *k* in our new subgraph.
- Now we look at graph G', for which we take another two connected subgraphs, which will have at least *maxflow* edges that are leaving the first of the subgraphs – however, since we are deleting F (a total of k edges) from this subgraph, then we must subtract this from the number of edges leaving the first subgraph, to get *maxflow* – *k* edges.
- So, we see that for this subgraph in our new graph G', a maximized s-t flow will generate *maxflow* – *k* edges, where the s-t flow will also be equal to, or greater than *maxflow* – *k.*

6. Given a sequence of numbers find a subsequence of alternating order, where the subsequence is as long as possible.

(that is, find a longest subsequence with alternate low and high elements).

As always, prove the correctness of your algorithm and analyze its time complexity.

We will utilize dynamic programming in order to store subsequences and generate the longest subsequence for which the numbers alternate. Here we will utilize a multidimensional array OPT[n][2], where n corresponds to the number of numbers given, and the columns correspond to two different scenarios. We can either have the scenario in which:

- The longest possible subsequence ends with an integer which is less than the integer before it – we will denote this as OPT[n][0]
- The longest possible subsequence ends with an integer which is more than the integer before it – we will denote this as OPT[n][1]

Next, we define the recurrence that will be used for the algorithm. We are seeking to maximize the length of the subsequence, so we will have the following recurrences:

Scenario 1:

- For j < i and the element OPT[i] < OPT[j], OPT[n][0] = the maximum of [OPT[i][0] and OPT[j][1] + 1]

Scenario 2:

- For j < i and the element OPT[i] > OPT[j], OPT[n][1] = the maximum of [OPT[i][1] and OPT[j][0] + 1]

In the first scenario, we end with an optimal solution whose last value is greater than the value right before it, so this subsequence of n values will either be equal to the OPT for which i is the last value, or the optimal solution OPT[j][1] + 1, where it *must* include the +1, because without it we would have a subsequence ending in a value which is less than that before it, but we want to add one which is greater than the value before it. We choose the maximum subsequence between the two. The converse of this is true for scenario 2, in which at position i, the value must be less than its previous element, so we choose the maximum subsequence between OPT[i][1] and OPT[j][0] + 1.

Time Complexity: The total time complexity of this algorithm will end up being $O(n^2)$, because for each of the *n* numbers, we are considering it in any one of the *n* positions in the sequence, and generating up to *n* possible positions that will result in an alternating order. Thereby, we have a total runtime of $O(n^2)$.