Assignment #4:

1. At any stage of an algorithm there will be three types of edges: E_1 is the set of pairs of nodes for which the the algorithm designer has asked the question and the answer given by the adversary was YES – these edges are already present at this stage. E_2 is the set of pairs of nodes for which the questions has already been asked and the answer given was NO – these edges are not present at this stage. $E_3 = E - E_1 - E_2$ is the set of pairs of nodes for which the question has not yet been asked and hence we have no knowledge about the existence or nonexistence of these edges at this stage. If there is spanning tree T such that $T \subseteq E_1$ then the algorithm designer needs to ask no more questions since (s)he knows that the graph G is connected. If on the other hand there is a partition of the nodes V into two sets V_1, V_2 such that $\{\{i,j\} : i \in V_1; j \in V_2\} \subseteq E_2$ – this means that there is a cut all of whose edges are contained in the set E_2, then also the algorithm designer needs to ask no more questions since (s)he knows that the graph G is not connected at this stage.

   Thus, any adversary strategy that needs to prolong an algorithm needs to make sure that either of these conditions do not occur until all questions have been asked. So when the algorithm designer checks if there is an edge between i and j, the adversary checks whether by saying YES if the new set E_1 has a spanning tree and if the answer is yes, the adversary says that this edge is not present. Similarly, if by answering No if this will create a cut, the adversary says that this edge exists. One way to achieve this is to have a default answer of NO unless this would create cut of nonexistent edges. You can do this by keeping a count that initially is $n-1$ at all nodes and decrease this by one each time a node is involved in a question and when the count reaches 1 say yes. You can also do by keeping a 0/1 matrix and achieve the same result. How we do this is irrelevant – it is the strategy that is relevant.

   This forces the algorithm designer to keep on guessing until the last question. Hence we force the algorithm designer to ask all questions which is $\frac{n(n-1)}{2}$.

2. 16.2-6: The following problem is known in the literature as the **knapsack problem:** We are given n objects each of which has a weight and a value. Suppose that the weight of object i is w_i and its value is v_i. We have a knapsack that can accommodate a total weight of W. We want to select a subset of the items that yields the maximum total value without exceeding the total weight limit.

   (i)If all v_i are equal, what would the greedy algorithm yield? Is this optimal?

   (ii)If all w_i are equal, what would the greedy algorithm yield? Is this optimal?

(iii) How should the greedy algorithm be designed in the general case? Is this optimal? [Be careful to distinguish between two versions of the problem: in one we are allowed to select fractional items and in the other we are not allowed to do this.]

Solution: [This a lot more than what I asked of you which is part (iii)fractional case]

In order to solve this problem most efficiently, we need the solution of Problem 9-2 which we do first;

- 9-2: Given n elements $x_1, x_2, ..., x_n$ with positive weights $w_1, w_2, ..., w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the weighted (lower) median is the element x_k that satisfies the relations:

$$\sum_{i:x_i<x_k} w_i < \frac{1}{2}$$

$$\sum_{i:x_i>x_k} w_i \leq \frac{1}{2}$$

We want to compute the weighted median in $\Theta(n)$ worst-case time.

Solution:

Find the regular median of the x_i. Let the index of of this element be i_1. Check the above two conditions with $k = i_1$. (Use PARTITION for this with this element as the pivot element). If they are satisfied, we have the required weighted median. If not either $\sum_{i:x_i<x_k} w_i \geq \frac{1}{2}$ or $\sum_{i:x_i>x_k} w_i > \frac{1}{2}$ (both can not happen – why?). In the first case, the weighted median is in the LOW side of the above PARTITION. In the second case, it is on the HIGH side. All elements on the other side can be set aside in the future steps. Since finding the median, PARTITION and condensing take $\Theta(n)$ worst-case time, the recurrence relation for this becomes:

$$t(n) = t(\left\lceil \frac{n}{2} \right\rceil) + \Theta(n)$$

whose solution is $t(n) = \Theta(n)$.

Back to Knapsack:

(i)We want an index k satisfying the relation:

$$\sum_{i:w_i<w_k} w_i \leq W$$

$$\sum_{i:w_i\leq w_k} w_i > W$$

We can do this in a manner similar to the above problem. Then by doing PARTITION on this element as the pivot, we can gather all elements on

the LOW side plus the fraction of this element equal to $\frac{W-\sum_{i\in LOW} w_i}{w_k}$. This takes in the worst-case $\Theta(n)$ time. This produces an optimal solution in this case.

Proof is similar to the proof for Activity Selection Problem done in class.

Suppose the optimal solution selects items in the set A and item 1 (in the increasing order of w_i) does not belong to A. If $A = \phi$, then $A \cup \{1\}$ is a better solution contradicting the optimality of A. If not let $k \in A$, and let $A' = A \cup \{1\} \backslash \{k\}$. A' contains item 1 and is also optimal. By induction, we get that the greedy solution is optimal.

(ii)We can do this in a manner similar to the above problem. Find the k^{th} largest element of the set of values in $\{v_i\}$ with $k = (\lfloor \frac{W}{w} \rfloor) + 1$ by SELECTION and do a PARTITION with this element as the pivot element. We can gather all elements on the HIGH side plus $\frac{W-kw}{w}$ fraction of k^{th}largest item. This takes in the worst-case $\Theta(n)$ time. This produces an optimal solution in this case. Proof is similar to the proof for (i)

(iii)Let $r_i = \frac{v_i}{w_i}$. Want to find an index k satisfying the relations:

$$\begin{aligned} \sum_{i:r_i > r_k} w_i &\leq W \\ \sum_{i:r_i \geq r_k} w_i &> W \end{aligned}$$

This is exactly like in Problem 9-2 (the value $\frac{1}{2}$ can be changed to any other value without loss). Now we do a PARTITION of the set of values $\{r_i\}$. The final solution includes all the HIGH side plus $\frac{W-\sum_{i\in HIGH} w_i}{w_k}$ fraction of item k. Proof is similar to (i). **This last part is for fractional solution**. The time taken is again $\Theta(n)$.

For the integer case: Do the same as above except if $0 < \frac{W-\sum_{i\in HIGH} w_i}{w_k} < 1$, remove from from further consideration the elements of the HIGH side and element k. Reduce W to $W - \sum_{i\in HIGH} w_i$ and repeat. But it may be better to sort all items in the first place; starting with item 1 (in this sorted order) include in the knapsack the next item if there is space in the knapsack; if not skip this item and go to the next item in the sorted order. However, **this algorithm does not always work** as shown by the following **example**:

	1	2	3
w	10	20	30
v	60	100	120

$W = 50$. Note that the items are already sorted as per the above algorithm. Hence, the greedy solution would be the set $\{1, 2\}$. The optimal solution is the set $\{2, 3\}$.

This algorithm is used as a heuristic in many instances. There is no known greedy algorithms that works for this problem. Indeed, there is no known polynomially bounded algorithm for this problem.

3. 16.1-3:

   Solution: Sort the activities in nondecreasing order of s_i. The complexity of this is $\Theta(n \lg n)$. Let the first activity be in lecture hall #1. There after, select the first activity not yet assigned a lecture hall and assign it a lecture hall that is already used (you can select the first created such a hall) if this does not create an overlap (conflict) and if it conflicts with activities already assigned in each of the halls used so far, create a new lecture hall and assign this activity to that hall. Clearly this is a "greedy" algorithm in its spirit. For the purpose of finding the lecture hall for the new activity, we keep track of the finish time of the last activity in each lecture hall used so far in the form of min-heap. We assign to new activity to the lecture whose "finish" time is the minimum. Every time an activity is assigned to a lecture hall that is already in use, its finish time is updated to the finish time of the activity now being assigned to that lecture hall and the heap is "fixed". This takes $\Theta(\lg k)$ time if k lecture halls are currently in use. When a new lecture hall is created to accommodate the current activity since it conflicts with every lecture hall in use, we need to add an element to the heap and insert it it in the right location. This also takes $\Theta(\lg k)$ time. This part of the algorithms takes $O(n \lg n)$ time for all activities put together.

   The reason why the algorithm "works" is that when we start a new lecture hall, there is an instant of time when there are as many activities that are active at that instant as there are lecture halls used. We need this many lecture halls if so many activities are simultaneously overlapping at this instant.

4. Consider the following problem: we have n boxes one of which contains the object we are looking for. The probability that the object is in the i^{th} box is p_i and this value is known at the outset. It costs c_i to look in the box i and these values are also known. The process is to look in some box; if the object is found, the process stops; if not, we look in some other box and so on till the object is found.

   Describe the greedy algorithm for this problem that works. What is the time complexity of the algorithm?

   To show that the algorithm works, you will need to answer the following:

   (a)If we look into box i first and don't find the object, what are the new probabilities for finding the object in box $j \neq i$? (These are called conditional probabilities).

   (b)Consider two scenarios: (I)First look in box i and if the object is not found, look in box j. (II) First look in box j and if the object is not found, look in box i. In each case, what is the probability of finding the

object in box $k \neq i$ or j, if we don't find the object in either box i or box j? Are they the same or are they different?

(c)Show that the algorithm works.

Solution:

We solve below the version in which we must look into the last box to retrieve the object. The case in which we do not have to look in the last box is a bit more complicated.

(o) Sort in increasing order of $\frac{c_i}{p_i}$. Look in this order until the object is found. Complexity $\Theta(n \lg n)$.

(a)Let $p = [p_1, p_2, ..., p_n]$. $p'_j = \frac{p_j}{1-p_i}$ for $j \neq i$.

(b)$p''_k = \frac{p_k}{1-p_i-p_j}$ for both cases.

(c)The cost of the search is given by the expression:

$$\begin{aligned} & c_{[1]} + (1 - p_{[1]})\{c_{[2]} + (1 - p'_{[2]})\{c_{[3]} + ... \\ = \quad & c_{[1]} + (1 - p\,[1])c_{[2]} + (1 - p_{[1]} - p_{[2]})c_{[3]} + ... \end{aligned}$$

where $[i]$ refers to the box searched at the i^{th} step if we have not found the object before; this depends on the algorithm. In the greedy algorithm, $[i]$ refers to the box that has the i^{th} smallest value of $\frac{c}{p}$ ratio. It is easy to show that if in the order [], we have two adjacent elements $i = [k]$ and $j = [k + 1]$ with $\frac{c_i}{p_i} > \frac{c_j}{p_j}$, then by interchanging these two we can reduce the total cost of the search. Hence, greedy algorithm works.