Assignment #4:

1. Partition V into two nearly equal size sets V_1 and V_2. For any pair (u, v) with both u, v in the same part of the partition, if asked, say that there is an edge. For all else till the last one say no. It is not possible to decide whether the graph is connected or not till the last question for which one end is in V_1 and the other is in V_2. So we need $\lceil \frac{n}{2} \rceil \lfloor \frac{n}{2} \rfloor = \Omega(n^2)$ questions.

2. Solution:

   (a) Consider three activities whose data is given below:
   #1: $[0, 8)$; #2: $[7, 10)$; #3: $[9, 20)$. For this problem, this proposed algorithm would first select activity #2 and would then be not able to select any others. But activities #1 and #3 are nonoverlapping. Hence this algorithm is not guaranteed to produce an optimal solution.

   (b) Consider three activities whose data is given below:
   #1: $[0, 25]$; #2: $[2, 5]$; #3: $[6, 10]$. For this problem, this algorithm selects #1 first and then is unable to select any more. The optimal answer is {2,3}. Hence this algorithm is not guanranteed to produce an optimal solution.

3. 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:

   (i)Include the items in increasing order of w_i until including the next item would make the load exceed the capacity of the knapsack. Include the next item (in increasing order of w_i) for the fraction that would bring the total weight equal to W. One way of doing this is to sort the items in increasing order of w_i. Find the largest value of k such that $\sum_{i=1}^{k} w_i \leq W$ when the items are sorted in the order mentioned. These are the items

selected to be included in the greedy solution for the integral case (For the fractional case, also include $\frac{W-\sum_{i=1}^{k} w_i}{w_{k+1}}$ fraction of item $k+1$). 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 above ordering) 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.

Implementation: There is a way to do this $O(n)$ time. Similar remarks apply to all parts for the fractional case. This needs another problem from the book:

Problem 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:

$$\begin{aligned} \sum_{i:x_i<x_k} w_i &< \frac{1}{2} \\ \sum_{i:x_i>x_k} w_i &\leq \frac{1}{2} \end{aligned}$$

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

Application to Knapsack:

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

$$\begin{aligned} \sum_{i:w_i<w_k} w_i &\leq W \\ \sum_{i:w_i\leq w_k} w_i &> W \end{aligned}$$

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.

(ii)Sort the items in decreasing order of v_i. Include items $1, 2, ..., k$ (where these are the indices of the items in this sorted order), where $k = \lfloor \frac{W}{w} \rfloor$ for the integer case. (For the fractional case, include also $\frac{W-kw}{w}$ fraction of item $k+1$). 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 above ordering) 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.

(iii)Sort items in increasing order of $\frac{w_i}{v_i}$. Let us suppose items are numbered in this order. The greedy algorithm for the integer case is as follows:

Starting with item 1 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.

For the fractional case, starting with item 1 include in the knapsack the next item if there is space in the knapsack. Suppose that there is no more space for item $k+1$ (and this is the first such item). Include $\frac{W - \sum_{i=1}^{k} w_i}{w_{k+1}}$ fraction of item $k+1$ and stop. This is an optimal solution to this problem under fractional item possibility.Consider the following generalization of a scheduling example done in class:

4. We have n customers to serve and m identical machines that can be used for this (such as tellers in a bank). The service time required by each customer is known in advance: customer i will require t_i time units ($1 \leq i \leq n$). We want to minimize $\sum_{i=1}^{n} C_i(S)$, where $C_i(S)$ represents the time at which customer i completes service in schedule S . How should the greedy algorithm work in this case?

Solution:

When $m = 1$, we have a single machine and this is the case we did in class. When we showed why it worked for this case, we derived the following formula for $\sum_{i=1}^{n} C_i(S)$:

Let $P = p_1 p_2 ... p_n$ be a permutation of $\{1, 2, ..., n\}$ and let $s_i = t_{p_i}$. If customers are served in the order corresponding to P,

$$\begin{aligned} \sum_{i=1}^{n} C_i &= s_1 + (s_1 + s_2) + ... + (s_1 + ...s_n) \\ &= \sum_{k=1}^{n} (n - k + 1) s_k \end{aligned}$$

When we have many machines, the number of jobs on any machine may depend on the schedule. For any machine, say i, we would get an expression similar to the one above:

$$\begin{aligned}\sum_{i=1}^{n} C_i &= s_1 + (s_1 + s_2) + ... + (s_1 + ...s_{n_i}) \\ &= \sum_{k=1}^{n_i} (n_i - k + 1)s_k\end{aligned}$$

In particular, the last job on any machine, has a multiplication factor equal to 1; the next to last job has a factor equal to 2 and so on. This is true for each machine. So in order to minimize $\sum_{i=1}^{n} C_i(S)$, we must ensure that the largest m jobs are placed at the end of the line in the m machines. Then, we place the next set of m largest jobs at the second last position one for each machine and so on. As an example, let $n = 15; m = 4; t = [12, 3, 8, 5, 10, 9, 6, 4, 13, 20, 16, 18, 7, 1, 5]$. Our algorithm would have the following evolution:

$\frac{positions \longrightarrow}{machines \downarrow}$	1	2	3	4
1		5	8	13
2	1	5	9	16
3	3	6	10	18
4	4	7	12	20

This is the same as assigning the jobs in increasing order one each to each of the machines in a round robin manner. This would yield the following:

$\frac{positions \longrightarrow}{machines \downarrow}$	1	2	3	4
1	1	5	9	16
2	3	6	10	18
3	4	7	12	20
4	5	8	13	

which is the same as the one above.

5. 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.

   (o)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.