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. So we need \( \left\lceil \frac{n}{2} \right\rceil \left\lfloor \frac{n}{2} \right\rfloor = \Omega(n^2) \) questions.

2. Exercise 16.2.7 (page 384):

Maximizing \( \prod_{i=1}^{n} [a_i]^{b_i} \) is equivalent to maximizing \( \{\log[\prod_{i=1}^{n} (a_i)^{b_i}]\} = \sum_{i=1}^{n} b_i \log(a_i)\} \).

Greedy Algorithm: Let the set \( \{b_i\} \) be sorted so that \( b_1 \leq b_2 \leq ... \leq b_n \). Sort the set \( \{a_i\} \) so that \( a_1 \leq a_2 \leq ... \leq a_n \). This is the required solution.

**Proof:** Suppose the algorithm does not work. Let \( S \) be an optimal solution in which we have a pair \((p, q)\) satisfying the relations

\[
\begin{align*}
& b_p < b_q \\
& a_p > a_q \\
\end{align*}
\]

Consider another solution \( S' \) in which we pair \( b_p \) with \( a_q \) and \( b_q \) with \( a_p \).

\[
\begin{align*}
\prod_{S} [(a_i)^{b_i}] : \prod_{S'} [(a_i)^{b_i}] \\
(a_p)^{b_p} (a_q)^{b_q} : (a_p)^{b_p} (a_q)^{b_q} \\
1 : \left( \frac{a_p}{a_q} \right)^{b_q-b_p} \\
\end{align*}
\]

Since \( a_p > a_q \) and \( b_q > b_p \), it follows that \( S' \) is a better solution contradicting the assumption that \( S \) was optimal.

3. Exercise 16.1-4:

(a) Consider four activities: \([1, 3); [1, 5); [6, 9); [4, 12)\). Algorithm that maximizes the number of nonoverlapping activities (as in the book which selects one with minimum finish time) selects activities \([1, 3)\) at the first sweep and since activities 2, 4 overlap, we need two more sets. The optimal solution is \([1, 4)\), and \([2, 3)\) requiring only two sets.

(b) Sort activities in increasing order of start times. Each time feed the next activity to the first set with which it does not overlap (if this means we must start a new set, do so). To prove that this is correct: we note that the number of sets needed is no less than \( k \) where \( k \) is largest number of overlaps at any point. This algorithm has exactly that number of sets since each time we start a new set, it is because this activity overlaps with one activity in each of the previous sets.

1
4. 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 \( x_1, x_2, \ldots, x_n \) with positive weights \( w_1, w_2, \ldots, w_n \) 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 cannot 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:

\[
t(n) = t\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + \Theta(n)
\]

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 $\frac{W - \sum_{i \in \text{LOW}} w_i}{w_k}$. 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 $A' = A \cup \{1\} \setminus \{k\}$. $A'$ 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 $\frac{W - kw}{w_k}$ 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:

$$\sum_{i: r_i > r_k} w_i \leq W$$
$$\sum_{i: r_i \geq r_k} w_i > W$$

This is exactly like in Problem 9-2 (the value $\frac{1}{w}$ 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 $\frac{W - \sum_{i \in \text{HIGH}} w_i}{w_k}$ 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 $0 < \frac{W - \sum_{i \in \text{HIGH}} w_i}{w_k} < 1$, remove from from further consideration the elements of the HIGH side and element $k$. Reduce $W$ to $W - \sum_{i \in \text{HIGH}} w_i$ 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:

\[
\begin{array}{c|c|c|c}
  & 1 & 2 & 3 \\
\hline
w & 10 & 20 & 30 \\
v & 60 & 100 & 120 \\
\end{array}
\]

\(W = 50\). 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.

5. 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 \(P = p_1p_2...p_n\) be a permutation of \(\{1, 2, ..., n\}\) and let \(s_i = t_{p_i}\). If customers are served in the order corresponding to \(P\),

\[
\sum_{i=1}^{n} C_i = s_1 + (s_1 + s_2) + ... + (s_1 + ... s_n)
\]

\[
= \sum_{k=1}^{n} (n - k + 1)s_k
\]

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:

\[
\sum_{i=1}^{n} C_i = s_1 + (s_1 + s_2) + ... + (s_1 + ... s_{n_i})
\]

\[
= \sum_{k=1}^{n_i} (n_i - k + 1)s_k
\]

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:

<table>
<thead>
<tr>
<th>positions $\rightarrow$ machines</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>positions $\rightarrow$ machines</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

which is the same as the one above.

6. 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 $\Theta(n \log n)$.

(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:


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.