Assignment 4

Jiadao Zou --- jxz172230

Q1

> 1. Let $G = [V, E]$ be an undirected graph. We want to check if it is connected. The only questions that we are allowed to ask are of the form: "Is there an edge between vertices i and j?". Using an adversary argument show that any correct deterministic algorithm to decide if G is connected must ask $\Omega(n^2)$ questions.

- First, we could use logic deduction
  - $\forall$ pair (m, n), there are totally three different situation:
    1. G_1, the set of pairs which have been asked and the adversary gave out "Yes" (exists an edge between the pair), already known.
    2. G_2, the set of pairs which have been asked and the adversary gave out "No" (no edge between the pair), already known.
    3. G_3, the set of pairs which have not been asked so we don't know whether they are connected, unknown.

- Second, the designer algorithms:

  1. Partition V into two parts, set V_1 and set V_2.
  2. $\forall$ pair (m, n), repeating:
     - If both m, n in the same set, then we assume there is an edge between those two vertices.
     - Else, m, n are not in the same set, assume there is no edge between them.
  3. Till out last question, "whether the end point in V_1 and the end point in V_2 are connected", we can't decide whether the graph is connected or not.

- Third, the adversary:

  - When the algorithms check if there is an edge between $m\ and\ n$:
    - Yes, if (m, n) are connected and that is within G_1 (connected graph).
      - The adversary will say the edge is not present.
    - No, if (m, n) are found to be not connected and there will be a cut.
      - The adversary will say the edge is exists. And the designer knows in this case the graph G is not connected and he needs to ask more questions.

  So we need to traverse all the cases $\frac{n(n-1)}{2} = O(n^2)$.

Q2

> 2. Exercise 16.2-7: Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i^{th} element of set A, and let b_i be the i^{th} element of the set B. You then receive a payoff of $\prod_{i=1}^{n}[(a_i)^{b_i}]$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

- We could sort A and B into monotonically increasing/decreasing order.
- Prove (Increasing Order):
  - Suppose we have indices $i,\ j$ that $i < j$. The corresponding a step of payoff is $(a_i)^{b_i},\ (a_j)^{b_j}$.
  - Since A and B are monotonically increasing sorted, $a_i \leq a_j$ and $b_i \leq b_j$. Still those two sets only contain positive values, thus $a_i^{b_j - b_i} \leq a_j^{b_j - b_i} \implies a_i^{b_j} a_j^{b_i} \leq a_i^{b_i} a_j^{b_j}$
  - We could see, this strategy give out a relatively larger $(a_i)^{b_i}(a_j)^{b_j}$ in each step.
- By the same idea, we could prove sorting in monotonically decreasing order gives out the same result.

Q3

> 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.]

- To avoiding exceeding the total load of Knapsack, we have to sort the items in increasing order of W. Then we only need to find the $k\ of \sum_{i=1}^{k} w_i \leq W$. For the fractional case, we have $\frac{W - \sum_{i=1}^{k} w_i}{w_{k+1}}$ for case $k+1$. There would produce a greedy optimal solution for both of

integral and fractional problem.
  - If the optimal solution has set J and object i which is not belongs to set J. If J is $\emptyset$, then $J \cup O_i$ is contradicting the optimal set J. Else, we have $O_k \in J$, and let $J' = J \cup O_i \setminus O_k$. J' is also optimal. Therefore, the greedy solution is optimal.

- Implement the algorithms in $O(n)$:
  - Referencing the previous question that:
    - Give 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:

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

    - We want to get the weighted median in $\Theta(n)$.
  - Q(i):
    - We want our index k satisfying the relation, just like above:

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

    - Doing PARTITION on this element as the pivot to get the LOW side elements. Also, the fraction of this element is $\frac{W-\sum_{i\in LOW} w_i}{w_k}$. This procedure is $\Theta(n)$ and produce the optimal solution.
  - Q(ii):
    - Sort the objects in decreasing order of v_i. Still we want to get $O_1, O_2 ... O_k$, where $k = \lfloor \frac{W}{w} \rfloor$ for integer case and $\frac{W-kw}{w}$ for fraction case of O_{k+1}.
    - The left is same with above question.
  - Q(iii): Algorithms design
    - First, sort objects in increasing order of $\frac{w_i}{v_i}$.
      - Integer Case:
        - Starting with O_1 and include the next object if that is still within the limit. If not, just skip it and go to the next object.
        - This algorithms doesn't work for all the time. There is no greedy method for the optimal solution.

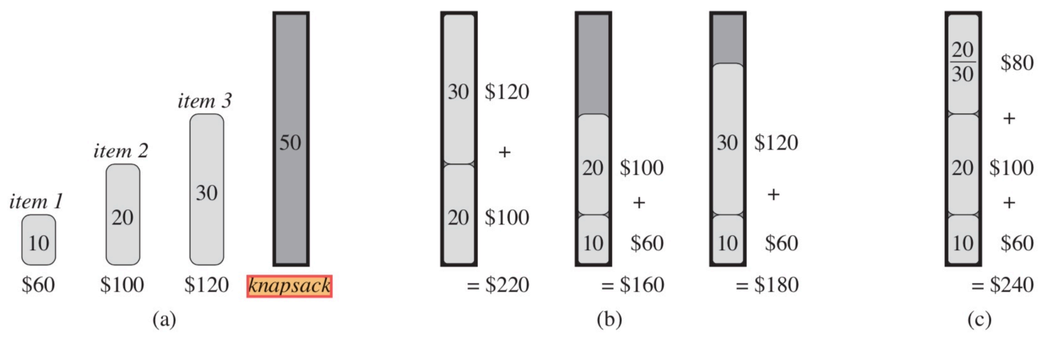


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

- Fractional Case:
  - By using $\frac{W-\sum_{i=1}^{k} w_i}{w_{k+1}}$ of O_{k+1}, there is an optimal solution.

Q4:

> 4. Consider the following generalization of a scheduling example done in class:
>
>   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? Is it guaranteed to produce optimal solutions?

- When $m = 1$:
  - 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 of P:

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

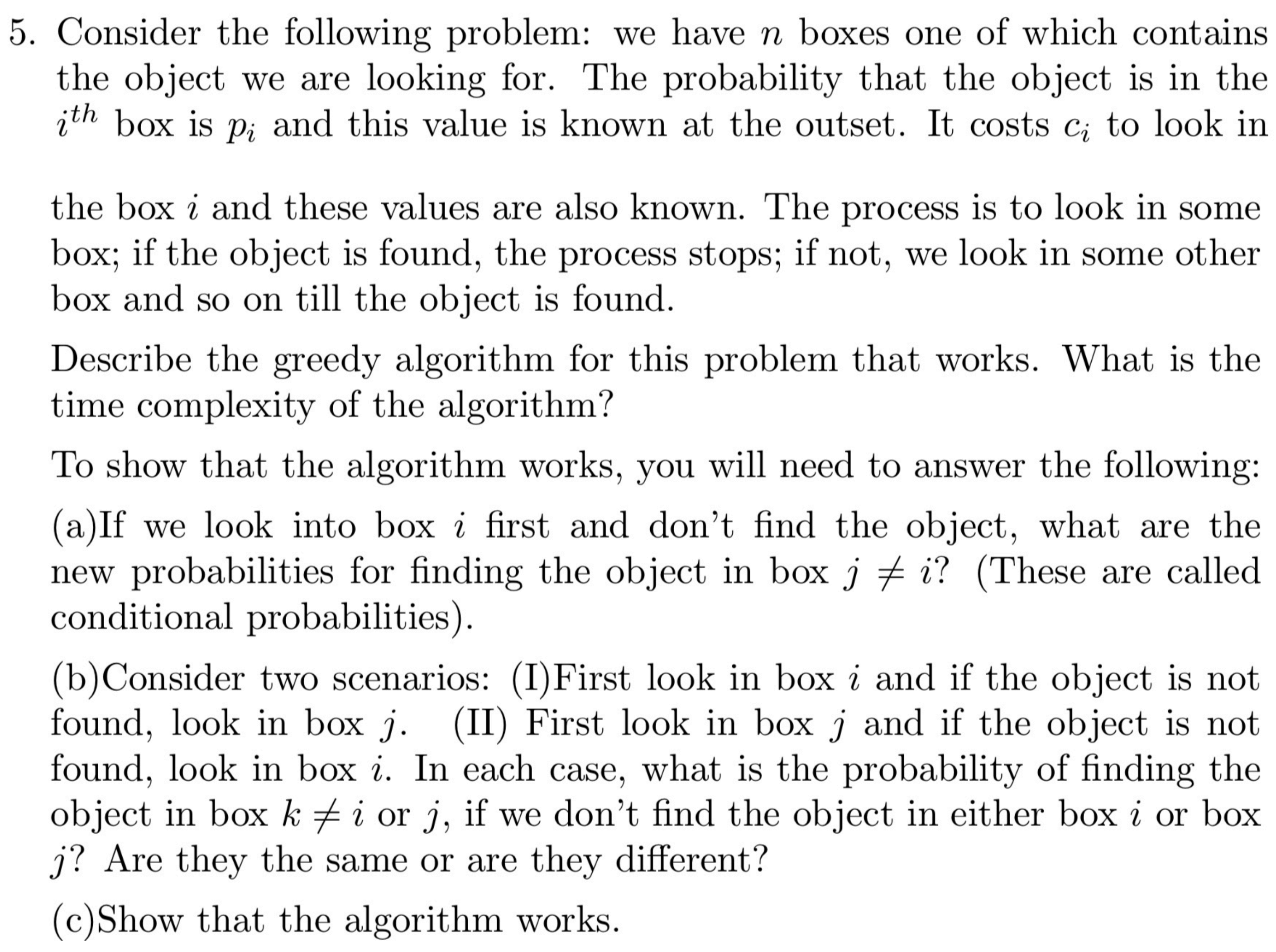
    This is minimized by permutation that $s_1 \leq s_2 ... \leq s_n$, the corresponding greedy algorithms is processing the customer with least time required first.

- When $m > 1$:
  - Just like above, for machine α, we still have the same relationship:

$$\sum_{i=1}^{n} C_i = s_1 + (s_1 + s_2) + \ldots + (s_1 + s_2 + \ldots + s_{n_\alpha})$$
$$= \sum_{k=1}^{n_\alpha} (n - k + 1) s_k$$

- Still the same, to minimize $\sum_{i=1}^{n} C_i$, for all m machines, we have to put the longest time required jobs at the end of each machine's schedule. The we put the remaining longest time required jobs at the second end of all m machines' schedule. And so on.

Q3



> 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.
>
> 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.

- Q(a):
  - $P_{j \neq i} = \frac{p_j}{1-p_i}$
- Q(b):
  - No matter the sequence of i and j, both cases are $P_{k \neq i,j} = \frac{p_k}{1-p_i-p_j}$.
- Q(c):
  - To maximize each step's benefit, we sort the box in increasing order of $\frac{c_i}{p_i}$, and search from the low side to high.
  - Algorithms Complexity:
    - Assume we totally take {1...k} steps:

$$\begin{aligned} Cost &= c_{(1)} + (1 - p_{(1)})(c_{(2)} + (1 - p_{not\ (1),(2)})(c_{(3)} + (1 - p_{not\ (1),(2),(3)})\dots) \\ &= c_{(1)} + (1 - p_{(1)})(c_{(2)} + (1 - p_{(1)} - p_{(2)})(c_{(3)} + (1 - p_{(1)} - p_{(2)} - p_{(3)})\dots) \end{aligned}$$

The (i) is the ith step till we found the object. It is also ith smallest relatively cost of $\frac{c_i}{p_i}$.

- The equation seems like $n!$, since we have fewer and fewer choices after each step. The complexity is $\Theta(n!) = \Theta(n \lg n)$