Bo Yang (Aaron) USCID: 7526922531

USC CSCI 570 Homework1 Date: 2021-02-09

1. 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 set B. You then receive a payoff on $\prod_{i=1}^{n} a_i^{b_i}$.

Give an algorithm that will maximize your payoff (6 points). Prove that your algorithm maximizes the payoff (10 points) and state its running time (4 points).

Solution:

> **My algorithm**:
>
> 1. sort A in a descreasing order.
> 2. sort B in a descreasing order.
> 3. do the $\prod_{i=1}^{n} a_i^{b_i}$.
>
> **Prrof**:
> I plan to use prove by contradiction.
> Assume there is an better optimal solution S' which can get a bigger payoff than mine S. That is to say, at least one pair of elements could have a different combination and get a greater result.
>
> Let's say, In S' solution, for some policy, a_1 will pair with b_j and a_i will pair with b_1.
>
> As for my solution S, a_1 paired with b_1 and a_i paired with b_j and the rest of pairs will be the same in both solution.
>
> Mention that $a_1 \geq a_i$ and $b_1 \geq b_j$, since we sorted data set in a descreasing order.
>
> $$S' \begin{matrix} a_1 & a_i & -------------------- \\ b_j & b_1 & -------------------- \end{matrix}$$
>
> $$S \begin{matrix} a_1 & a_i & -------------------- \\ b_1 & b_j & -------------------- \end{matrix}$$
>
> So the payoff of S' is:
> $Payoff(S') = \prod_{(i,j)\in S'} a_i^{b_i}$ ①
> $Payoff(S) = \prod_{(i,j)\in S} a_i^{b_i}$ ②
>
> $$\begin{aligned} \frac{①}{②} &= \frac{{a_1}^{b_j} \cdot {a_i}^{b_1} \cdot rest}{{a_1}^{b_1} \cdot {a_i}^{b_j} \cdot rest} \\ &= \frac{{a_i}^{b_1 - b_j}}{{a_1}^{b_1 - b_j}} \\ &= \left(\frac{a_i}{a_1}\right)^{b_1 - b_j} \end{aligned}$$
>
> Since $a_1 \geq a_i$ and $b_1 \geq b_j$
> $so \frac{①}{②} \leq 1 => Payoff(S') < Payoff(S)$
> The result contradicts the assumption that S' is the optimal solution.
>
> **Runtime**:
> Obviously, my algorithm will cost $O(n)$ time if two data set already sorted before.
> If the data set not sorted, we need to spend at least $O(n \cdot log(n))$ time to sort first.

If the data set not sorted, we need to spend at least $O(n \cdot log(n))$ time to sort first, and then do the algorithm. In this case, my algorithm will cost $O(n \cdot log(n))$ time.

2. Suppose you were to drive from USC to Santa Monica along I-10. Your gas tank, when full, holds enough gas to go p miles, and you have a map that contains the information on the distances between gas stations along the route. Let $d_1 < d_2 < \ldots < d_n$ be the locations of all the gas stations along the route where d_i is the distance from USC to the gas station. We assume that the distance between neighboring gas stations is at most p miles. Your goal is to make as few gas stops as possible along the way.

Give the most efficient algorithm to determine at which gas stations you should stop (6 points) and prove that your strategy yields an optimal solution (10 points).
Give the time complexity of your algorithm as a function of n (4 points). Suppose the gas stations have already been sorted by their distances to USC.

Solution:

I'm going to use greedy algorithm to solve this problem.

My algorithm:

1. Based on distances array D, compute intervals array I. (interval between each gas station)
2. At each gas station, check if you can reach to next station without stopping.
3. if you can, then go. (update the remaining_distance variable)
4. if not, stop += 1 and fill the tank. (update the remaining_distance variable)
5. keep doing step 2,3,4 until reach the destination.

Prrof: I plan to use prove by contradiction.
Assume there is a better optimal solution S' which can get smaller stops than mine S. That is to say, S' has a better policy and is able to stop fewer times, but still can arrive at the end.

Let's say, In my solution S, I chose the i_{th} station as my first stop. $(i < n)$ And keep following the same policy in the remaining $(n - i)$ stations. Finally, my algorithm needs to stop s times to get to Santa Monica.

if there were a better solution S' could solve this problem with at most $s - 1$ stops. This better solution S' should have chosen j_{th} stop as its first stop $(i < j)$, but the maximum distance the car can run is fixed, implying that there is no way for this algorithm to reach j_{th} station as it first stop station. So it either reaches i_{th} station or reaches a station which is ahead of i_{th} station. Then this algorithm will stop more than s time in the end, which is contradict our assumption.

PsedoCode:

```
p = 5
    dis = [1, 3, 8, 11, 15, 18, 20, 21]
    intervals = [b - a for a, b in zip(dis[:-1], dis[1:])]
    intervals.insert(0, dis[0])
    print(intervals) # [1, 2, 5, 3, 4, 3, 2, 1]

    remain = p
    dp = [0] * (len(dis) + 1)

    for i in range(1, len(dis) + 1):
        if remain < intervals[i - 1]:
            dp[i] = dp[i - 1] + 1
            remain = p
        else:
            dp[i] = dp[i - 1]
            remain -= intervals[i - 1]

    print(dp[1:])
```

Runtime:
Obviously, my algorithm will cost $O(n)$ time.

3. Some of your friends have gotten into the burgeoning field of time-series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges-what's being bought-are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them---for example, they may want to know if the four events

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible events *(e.g., the possible transactions)* and a sequence S of n of these events. A given event may occur multiple times in S *(e.g., Yahoo stock may be bought many times in a single sequence S)*. We will say that a sequence S' is a subsequence of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you:
Give an algorithm that takes two sequences of events-S' of length m and S of length n, each possibly containing an event more than once-and decides in time $O(m+n)$ whether S' is a subsequence of S. Prove that your algorithm outputs "yes" if S' is indeed a subsequence of S (hint: induction).

Solution:

I'm going to use two pointers to solve this problem.

My algorithm:

1. One pointer point to the S, the longer sequence.
2. Another one pointer point to S', the shorter sequence.
3. both pointers point to next position if the elements they pointed are same. (update an variable $length$)
4. if not, pointer of S continue move to next position, and the pointer of S' stay there.
5. after iteration both sequences, check if the variable $length$ equals to the length of S'. if true, return yes

Prrof:
As it asked, I need to use induction method to prove.
I use p to denote the current position in S, and q the current position in S'.
And use $k_1, k_2, ..k_r$ to denote the matched event.
And assume S' is the subsequence $s_{loc_1}, s_{loc_2}, ..s_{loc_r}$ of S.
In other word, our algorithm could find a matching k_q and $k_q \leq loc_q$. s.t. q = 1, 2...,m.

Base case: when $q = 1$. The first matching event k_1 is same as s'_1. This means $k_1 \leq loc_1$.

Inductive hypothesis: Assume when $q - 1 < m$, The macthing event $k_{q-1} \leq loc_{q-1}$.

Inductive step: Prove the event k_q is also less or equal to loc_q if it exists in S'.

Since the loc_q is the final matching event in S', so it must has $loc_q > loc_{q-1}$.
And based on *inductive hypothesis*, we can get $loc_{q-1} \geq k_{q-1}$. So we have $loc_q > k_{q-1}$.
Also need to mention that the greedy algorithm will find the matching if they are equal. So $k_q \leq loc_q$.

Runtime:
Obviously, my algorithm will cost $O(m+n)$ time.

4. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

Solution:

> Let me rephrase the question. There is n packages and each package i has a weight w_i. Also, we use m trucks t whose maximum capacity is W to transport these packages. Assume the greedy algorithm S provided in the question is able to use m to finish this task. If this greedy solution S is non-optimal, then there might exist a better algorithm S' can solve this problem with less trucks, implying that $S(n) = m$ and $S'(n) = k$ and $m \geq k => S(n) \geq S'(n)$
>
> > So **greedy algorithm** S use $t_1, t_2, \dots t_m$ to do transportion.
> > And **optimal algorithm** S' use $t_1, t_2, \dots t_k$ to do transportion.
> > The goal is to prove $m = k$. and we use contradiction to prove.
> >
> > 1. First, assume $k > m$, if $k > m$ then S' is not the optimal solution anymore, since the S could use less trucks to solve problem.
> > 2. Then assume $k < m$, which in solution S', there exists some trucks load more packages than solution S. Only in this way could the S' sent less truck but still can transport same amount of goods. But the question said that the solution S "pack boxes in the order boxes arrive, and whenever the next box does not fit, they send the truck on its way". This mean solution trys so hard and ensure every truck cannot load anymore goods. So in the solution S', k not less thans m.
> > 3. So $k = m$, k must equal to m. and the greedy solution S must be the optimal.

5. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that $T(\cdot)$ represents the running time of an algorithm, i.e. $T(n)$ is positive and non-decreasing function of n and for small constants c independent of n, $T(c)$ is also a constant independent of n. Note that some of these recurrences might be a little challenging to think about at first. Each question has 4 points. For each question, you need to explain how the Master Theorem is applied (2 points) and state your answer (2 points).

- (a) $T(n) = 4 \cdot T(n/2) + n^2 \cdot log(n)$
- (b) $T(n) = 8 \cdot T(n/6) + n \cdot log(n)$
- (c) $T(n) = \sqrt{6006} \cdot T(n/2) + n^{\sqrt{6006}}$
- (d) $T(n) = 10 \cdot T(n/2) + 2^n$
- (e) $T(n) = 2 \cdot T(\sqrt{n}) + log_2 n$

Solution:

> for question **(a)**:

> $a = 4, b = 2, c = log_b a = 2, f(n) = n^2 \cdot log(n), f(n) = \Theta(n^2 \cdot log(n))\ and\ k = 1$
> $since\ k = 1 \geq 0,\ then\ T(n) = \Theta(n^2 \cdot (log(n))^2)$

for question **(b)**:

> $a = 8, b = 6, c = log_b a = log_6 8 > 1, f(n) = n \cdot log(n), f(n) = O(n^{log_6 8 - \epsilon})$
> $then\ T(n) = \Theta(n^{log_6 8})\ for\ some\ 0 < \epsilon < log_6 8 - 1$

for question **(c)**:

> $a = \sqrt{6006}, b = 2, c = log_b a = log_2 \sqrt{6006} > 1, f(n) = n^{\sqrt{6006}}, f(n) = \Omega(n^{log_2 \sqrt{6006}}).$
> $then\ T(n) = \Theta(n^{\sqrt{6006}}) \quad for\ some\ \epsilon > 0$

for question **(d)**:

> $a = 10, b = 2, c = log_b a = log_2 10 > 1, f(n) = 2^n, f(n) = \Omega(n^{c+\epsilon})$
> $then\ T(n) = \Theta(2^n) \quad for\ any\ \epsilon > 0$

for question **(e)**:

> Assume $n = 2^m, then\ T(2^m) = 2 \cdot T(2^{m/2}) + m$ And change the form again.
> $T(2^m) = F(m)$, so we have $F(m) = 2 \cdot F(m/2) + m$
> In this case, $a = 2, b = 2, c = log_b a = 1, f(m) = m$
> $then\ F(m) = \Theta(m \cdot log(m))$