CSCI 570

Analysis of Algorithms - Assignment II

Harshitha Kurra | USC ID: 3161505044

---

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 the set A, 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.

Prove that your algorithm maximizes your payoff and state its running time.

Solution:

Algorithm :

Step 1: sort set A. We are using quick sort algorithm for the best time complexity.

PseudoCode:

```
def partition (A[ ], low, high):
   // pivot (Element to be placed at right position)
   set pivot to the last element of the array
   i = (low - 1)  // Index of smaller element
   for (j = low; j <= high - 1 ; j - -):
        if current element is lower than the pivot then:
                #increment the index of smaller element and swap the elements(current
                element and smaller element)

   swap A[i + 1] and A[high]
   return (i + 1)
```

Step 2: sort set B - same as Step 1

Step 3: return $\prod_{i=1}^{n} a_i^{b_i}$ ==> a_i should be paired with b_i

$\underline{Proofofcorrectness}$: Let set a_i be sorted so that $a_1 \leq a_2 \leq a_3 \leq \ldots\ldots. \leq a_k$ and b_i be sorted so that $b_1 \leq b_2 \leq b_3 \leq \ldots\ldots\ldots \leq b_l$. Pairing a_i and b_i is the optimal solution S.

By **Proof of contradiction**, Let the above provided solution S where a_1 is paired with b_1 and a_k is paired with b_l is not optimal (i.e; first element of set A with the first element of set B, second element of set A with the second element of set B etc.). Let us consider a solution S^1 where a_1 is paired with b_l and a_k is paired with b_1 where $a_1 < a_k$ and $b_1 < b_l$ (first element of set A with the last element of set B, second element of set A with second to the last element of set B etc.)

Then,

$$\frac{PayOff(S^1)}{PayOff(S)} = \frac{\prod_{S^1} a_i^{b_i}}{\prod_S a_i^{b_i}}$$

$$= \frac{(a_1)^{b_l}(a_k)^{b_1}}{(a_1)^{b_1}(a_k)^{b_l}}$$

$$= (\frac{a_1}{a_k})^{b_l - b_1}$$

≤ 1 since $a_k \geq a_1$ and $b_l \geq b_1$

Hence, by proof of contradiction, a_1 should be paired with b_1, a_2 should be paired with b_2 etc. for the remaining elements.

$\underline{Running time:}$

If the two sets A and B are already sorted then it takes $O(n)$ time to pair them but if we have to sort them first, as we use the efficient sort algorithm, it takes $O(nlogn)$

---

2. **Suppose you are 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 $d1 < d2 < \ldots < dn$ be the locations of all the gas stations along the route where d_i is the distance from USC to the i^{th} 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 and prove that your strategy yields an optimal solution. Give the time complexity of your algorithm as a function of n. Suppose the gas stations have already been sorted by their distances to USC.**

Solution:

$\underline{Algorithm:}$

The approach that we follow here is greedy. We start at a point and assign the start point to current position. In the while loop, we check if my current position is not the end. If it is not, we check if the end is more than p miles from now. If it is, we get the distance of the closest gas station before reaching the end(the last gas station which is in p miles distance from current location). And set our current position to that position. We check so on.. till we reach the destination.

```
def AlgoForLessGasStops:
    current <— start
    while(current != end) :
        now <— Distance from current where the car will run out of gas
    ############# greedy ########################
        if(now < end):
            position_of_gas_station = gas station before reaching the end
            current = position_of_gas_station
        else:
            current = end
```

Proofofcorrectness :

Let A be the output set produced by the algorithm. The question is whether there exists any other solution set B that is better than A. Implies, B is optimal. If B exists, then for it to be optimal, the number of elements in B is less than or equal to A.

$$\begin{tikzcd} A = A[1], A[2], A[3] \ldots . A[n] \arrow[d, "\textbf{Replace } B[1] \textbf{ with } A[1]"] \\ B = B[1], B[2], B[3], \ldots . B[k] \end{tikzcd}$$

Consider B. If it includes A [1] (the first element of A), then the greedy algorithm worked at the first step. If B did not have A [1], then let us remove B [1] (the first element of B)

and replace it with A [1]. The set B is still of the same size. At this point B is still feasible since B is optimal, the distance of A[1] is at least as much distance from the destination as B[1]. So, the replacement of B [1] with A [1] is still feasible.
The problem is now to find optimal solution from A [1] to Santa Monica. If B is the optimal set, then, B – A [1] is also optimal for the sub problem. This is because, if there were a smaller solution set C for traveling from A [1] to Santa Monica, then
C+A [1] will be smaller than B. But, C is impossible, because B is the optimal set.
Hence by **proof of contradiction**, the solution set A exactly matches with the optimal solution B.
Thus, the solution A found by the greedy algorithm is optimal.

$\underline{TimeComplexity}$:
We have the below loop in the algorithm.

```
while (condition):
.
.
.
```

The while loop runs for n times. Therefore, the algorithm can compute min stops with the time complexity of $O(n)$ where n is the number of gas stations.

---

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:

PseudoCode

```
Subsequence(){
set i and j to 1
when i <= S.length & j <= S'.length{
  if S(i) is the same as S'(j)
   then increment i, j by 1
   else just increment i
}
if ( j == S'.length) {
   return "Yes"
else
   Return "No"
```

<u>$TimeComplexity:$</u>

The while loop runs for the size of the subsequence limit i.e; m. One iteration takes $O(1)$. but the worst case scenario is we need to search the entire sequence to check if we could find the subsequence i.e; n so the total complexity will be

$O(m + n(1))$

$= O(m + n)$

<u>$Proof:$</u>

Assume the subsequence of S is $s_{k_1}, s_{k_2}, s_{k_3} \ldots\ldots s_{k_m}$ which is same as S^1 and the through the algorithm we find a subsequence/match, let it be $p_1, p_2, p_3, \ldots\ldots p_m$ such that $p_j \leq k_j$ where j= 1,2,3,…m. By **<u>proof of Induction</u>**,

<u>Base Case:</u> Consider the case where $j = 1$, then the algorithm let p_1 be the first occurrence which is same as $s_1{}^1$ and hence $p_1 \leq k_1$.

<u>Inductive Hypothesis and Inductive Step:</u> Now the case $j > 1$, assume $j - 1 < m$ and we found a match p_{j-1} where $p_{j-1} \leq k_{j-1}$. p_j will be the first event after p_{j-1} which is same as $s_j{}^1$ if an event exists. We know that k_j is such an event and $k_j > k_{j-1}$

And we know $k_{j-1} \geq p_{j-1}$

Therefore, $s_{k_j} = s_j{}^1$ and $k_j > p_{j-1}$

The algorithm finds the first such index so we get $p_j \leq k_j$. Hence Proved

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:

Consider the solution given by the greedy algorithm as a sequence of packages, represented by indexes: 1, 2, 3, ... n. Each package b_i has a weight, w_i, and the max weight a truck can carry is W. Let there be total of T trucks to carry the boxes.

Conditions: truck should not be overloaded i.e $\sum_{i=1}^{n} w_i \leq W$ and the order of boxes should be preserved by the time they arrive at the company. i.e; if b_i is sent before b_j in different trucks, b_i should arrive before b_j to the company. $(i < j)$.

If the greedy algorithm fits boxes $b_1, b_2, b_3, \ldots . b_j$ into k trucks and lets say other solution fits $b_1, b_2, b_3, \ldots . b_i$ into k trucks for $i \leq j$

By **proof of induction**,

Base Case: consider k=1 (the number of trucks is 1). The greedy algorithm indeed tries to fit in as many boxes as possible into the first truck. So it is true for k = 1

Inductive Hypothesis: assuming $k - 1$ holds, greedy algorithm fits m boxes into the first $k - 1$ trucks and let's say other solution fits m^1 such that $m^1 \leq m$.

Inductive Step: Now for the k^{th} truck, we need to pack $b_{m^1+1}, \ldots \ldots . b_i$ for the alternate solution and as we know $m^1 \leq m$, the greedy algorithm is at least able to fit in all the boxes $b_{m+1} \ldots b_i$ into the k^{th} truck and probably more.

Hence Proved.