ShanghaiTech University

CS240 Algorithm Design and Analysis

Spring 2023

Problem Set 2

Due: 23:59, March 17, 2023

1. Submit your solutions to the course Blackboard.
2. If you want to submit a handwritten version, scan it clearly.
3. Late homeworks submitted within 24 hours of the due date will be marked down 25%. Homeworks submitted more than 24 hours after the due date will not be accepted unless there is a valid reason, such as a medical or family emergency.
4. You are required to follow ShanghaiTech's academic honesty policies. You are allowed to discuss problems with other students, but you must write up your solutions by yourselves. You are not allowed to copy materials from other students or from online or published resources. Violating academic honesty can result in serious penalties.

Problem 1:

Consider a rural road (assumed to be a long line segment with a left and right endpoint) with a set of houses located along the road. Unfortunately, there is no access to piped water in this area and the residents of the houses can only get water by digging wells. As the community leader, you want to dig wells at certain locations on the road so that each house is within at most four miles from a well.

Give an efficient algorithm to achieve this goal while minimizing the number of wells to be dug, and prove the correctness of your algorithm.

Solution:

Here's a greedy algorithm for this problem. Start at the western end of the road and begin moving east until the first moment when there's a house h exactly four miles to the west. We place a well at this point (if we went any farther east without placing a well, we wouldn't cover h). We then delete all the houses covered by this well, and iterate this process on the remaining houses.

Here's another way to view this algorithm. For any point on the road, define its position to be the number of miles it is from the western end. We place the first well at the easternmost (i.e. largest) position s_1 with the property that all houses between 0 and s_1 will be covered by s_1. In general, having placed $s_1, \ldots, s_i$, we place well $i+1$ at the largest position S_{i+1} with the property that all houses between S_i and S_{i+1} will be covered by s_i and S_{i+1}.

Let $S = \{s_1, \ldots, s_k\}$ denote the full set of well positions that our greedy algorithm places, and let $T = \{t_1, \ldots, t_m\}$ denote the set of well positions in an optimal solution, sorted in increasing order (i.e. from west to east). We must show that $k = m$.

We do this by showing a sense in which our greedy solution S "stays ahead" of the optimal solution T. Specifically, we claim that $s_i \geq t_i$ for each i, and prove this by induction. The claim is true for $i = 1$, since we go as far as possible to the east before placing the first well. Assume now it is true for some value $i > 1$; this means that our algorithm's first i centers $s_1, \ldots, s_i$ cover all the houses covered by the first i centers $t_1, \ldots, t_i$. As a result, if we add t_{i+1} to $s_1, \ldots, s_i$, we will not leave any house between s_i and t_{i+1} uncovered. But the $(i+1)$st step of the greedy algorithm chooses S_{i+1} to be as large as possible subject to the condition of covering all houses between s_i and S_{i+1}; so we have $S_{i+1} \geq t_{i+1}$. This proves the claim by induction.

Finally, if $k > m$, then $s_1, \ldots, s_m$ fails to cover all houses. But $S_m \geq t_m$, and so $t_1, \ldots, t_m = T$ also fails to cover all houses, a contradiction.

Problem 2:

Suppose you have a single machine on which you want to schedule a set of jobs for n customers. You want to find a schedule to finish all the jobs which will make the customers the happiest. Let t_i denote the length of customer i's job, and let C_i be the job's finishing time in the schedule. For example, if jobs i and j are the first and second jobs in the schedule, then $C_i = t_i$, and $C_j = t_i + t_j$. Each customer i also has a weight w_i representing her job's importance, and her happiness is determined by both her job's importance and its finishing time. Specifically, you want to order the jobs to minimize the weighted sums of the jobs' completion times, defined as $\sum_{i=1}^{n} w_i C_i$. Design an efficient algorithm for this task, and prove that your algorithm is correct.

As an example, suppose there are two jobs, with $t_1 = 1$ and $w_1 = 100$, and $t_2 = 5$ and $w_2 = 4$. Doing the jobs in the order 1, 2 yields a weighted completion time of $100 \times 1 + 4 \times (1 + 5) = 124$, while doing them in the order 2, 1 gives a weight completion time of $100 \times (1 + 5) + 4 \times 5 = 620$. Thus, your algorithm should output the ordering 1, 2.

Solution:

An optimal algorithm is to schedule the jobs in decreasing order of w_i/t_i. We prove that this algorithm is optimal by an exchange argument.

Consider any other schedule. As is standard in exchange argument, we observe that this schedule must contain an inversion:

A pair of jobs i, j, for which i comes before j in the alternation solution, and j comes before i in the greedy algorithm. For this pair, we have $w_j/t_j \geq w_i/t_i$, by the definition of the greedy schedule.

We can show that swapping this pair i, j does not increase the weighted sum of completion times, which means we need to prove:

$w_j(t_i + t_j) + w_i t_i \geq w_i(t_i + tj) + w_j t_j$

That is:

$w_j t_i \geq w_i t_j$

Which is exactly held by the definition of the greedy schedule.

Then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having increased the function we are trying to minimize proves that the greedy algorithm is optimal.

Problem 3:

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 b_i be the i'th element of 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.

Solution:

Sort A and B into monotonically decreasing order. Here's a proof that this method yields an optimal solution. Consider any indices i and j such that $i < j$, and consider the terms $a_i^{b_i}$ and $a_j^{b_j}$. We want to show that it is no worse to include these terms in the payoff than to include $a_i^{b_j}$ and ${a_j}^{b_i}$, i.e., that ${a_i}^{b_i} a_j b_j \geq {a_i}^{b_j} {a_j}^{b_i}$. Since A and B are sorted into monotonically decreasing order and $i < j$, we have $a_i \geq a_j$ and $b_i \geq b_j$. Since a_i and a_j are positive and $b_i - b_j$ is nonnegative, we have $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$. Multiplying both sides by $a_i^{b_j} a_j^{b_j}$ yields $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.

Since the order of multiplication doesn't matter, sorting A and B into monotonically increasing order works as well.

Problem 4:

We are given a gray-scale picture consisting of an $m \times n$ array A of pixels, where each pixel specifies a single intensity value. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns. A 'seam' is defined as the set of removed pixels.

(a) Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

(b) Suppose that for each pixel $A[i, j]$, we have calculated a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure is, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels. Find the best seam, i.e. the seam with the smallest disruption measure.

Solution:

(a) If $n > 1$, then for every choice of pixel at a given row, we have at least 2 choices of pixel in the next row to add to the seam (3 if were not in column 1 or n). Thus the total number of possibilities is bounded below by 2^m.

(b) Use DP.

Initialization: Create a table $D[1 : m, 1 : n]$ such that $D[i, j]$ stores the disruption of an optimal seam ending at position $[i, j]$, which started in row 1. $D[1, j] = d[1, j]$ for $i = 1$ to n.

Transfer function:

For $i = 2$ to m, $j = 1$ to n

$$D[i,j] = \begin{cases} d[i,j] + \min(D[i-1,j], D[i-1,j+1]) & \text{if } j = 1 \\ d[i,j] + \min(D[i-1,j-1], D[i-1,j]) & \text{if } j = n \\ d[i,j] + \min(D[i-1,j-1], D[i-1,j], D[i-1,j+1]) & \text{otherwise} \end{cases}$$

Result: $\min(D[m, 1...n])$.

Time complexity of the algorithm is $O(mn)$.

Problem 5:

Jack wants to invite his friend John to have a meal in a restaurant, but Jack only has M yuan of money. The restaurant has n different kinds of dishes. The i'th dish costs a_i yuan. John is picky and doesn't want to eat the same dish more than once. John is also greedy, and wants to use up all of Jack's money. John wants to know how many ways there are to order dishes so that each dish is ordered at most once, and the total price of the dishes adds up to M. For example, if there are 4 dishes with price $\{1, 2, 3, 4\}$ and Jack has 6 yuan, then there are two ways John can order, namely the dishes with costs 1, 2, 3, or with costs 2, 4, and so the algorithm should output 2.

Design an efficient algorithm which takes Jack's money M and the prices of n dishes as input, and outputs the number of the ways to order the food which satisfy John's requirements. Analyze the complexity of your algorithm.

Solution:

There should be 3 possible outcome when we have m money left and consider the dish i, which is $f(m, i)$:

1. If we exactly have enough money for this dish, which is $m = a_i$, bought it will be a possible order, so we remove dish i in the remained dishes and add one possible order. $f(m, i) = f(m, i-1) + 1$

2. If we have more money than this dish's price, which is $m > a_i$, there will be two possible orders: to buy or not to buy. $f(m, i) = f(m, i-1) + f(m-a_j, i-1)$.

3. If we don't have enough money, the only possibility is to give up this dish. $f(m, i) = f(m, i-1)$

It is both possible for you to build a $M \times n$ table to save the possible order to each situation in this table or just build the initial case and run $f(M, n)$. Here we only give the code for the previous solution:

```
#The 0'th row and 0'th column is not used in the building table.
f=[[0 for i in range(M+2)] for j in range(n+2)]
for i in range(1,n+1):
    for j in range(1,M+1):
        if j==a[i]:                 #Situation 1
            f[i][j]=f[i-1][j]+1
        elif j>a[i]:                #Situation 2
            f[i][j]=f[i-1][j]+f[i-1][j-a[i]]
        else:                       #Situation 3
            f[i][j]=f[i-1][j]
```

return f[M][n]

Both time complexity and space complexity should be $O(Mn)$.

Problem 6:

Given two strings s and t, an *interleaving* of s and t is a string formed by first dividing s and t into n and m substrings respectively, for n and m differing by at most 1, and then concatenating the substrings in an alternating fashion. In other words, let $\|$ denote the concatenation operation, and write $s = s_1 \| s_2 \| ... \| s_n$ and $t = t_1 \| t_2 \| ... \| t_m$, for some strings $s_1, ..., s_n, t_1, ..., t_m$ and $|n - m| \leq 1$. Then the interleaving of s and t is any of the strings $s_1 \| t_1 \| ... \| s_n \| t_m$, or $s_1 \| t_1 \| ... \| t_m \| s_n$, or $t_1 \| s_1 \| ... \| t_m \| s_n$ or $t_1 \| s_1 \| ... \| s_n \| t_m$, depending on the sizes of n and m. For example, "hodtog" and "doghot" are both interleavings of the words "hot" and "dog".

Suppose you are given three strings r, s and t. Give an algorithm to determine if r can be formed by some interleaving of s and t. Analyze the complexity of your algorithm.

Solution:

First of all, if $|s_1| + |s_2| \neq |s_3|$, then s_3 must not be composed of s_1 and s_2 interleaved. When $|s_1| + |s_2| = |s_3|$, we can solve it by dynamic programming. We define whether $f(i, j)$ denotes the first i elements of s_1 and the first j elements of s_2 can intersect the first $i + j$ elements that make up s_3. If the first i element of s_1 is equal to the $i + j$ element of s_3, so whether the first i elements of s_1 and the first j elements of s_2 can intersect to form the first $i + j$ elements of s_3 depends on whether the first $i - 1$ elements of s and the first j elements of s_2 can intersect to form the first $i + j - 1$ elements of s_3, that is, $f(i, j)$ depends on $f(i - 1, j)$, in which case if $f(i - 1, j)$is true, then $f(i, j)$ is also true. Similarly, if the j element of s_2 is equal to the $i + j$ element of s_3 and $f(i, j - 1)$ is true, then $f(i, j)$ is also true:

$$f(i, j) = [f(i - 1, j) \text{ and } s_1(i - 1) = s_3(p)] \text{ or } [f(i, j - 1) \text{ and } s_2(j - 1) = s_3(p)]$$

Where $p = i + j - 1$ .The boundary condition is $f(0, 0) = true$