Kushagr,Gupta & Rayan,Dasoriya
kgupta7,rdasori@ncsu.edu

1. [5 points for part (a); 4 points for (b); 1 point each for (c) and (d)]

   Purpose: Applying dynamic programming to a new problem.

   Suppose n jobs, numbered $1, \ldots, n$ are to be scheduled on a single machine. Job j takes t_j time units, has an integer deadline d_j, and a profit p_j. The machine can only process one job at a time and each job must be processed without interruption (for its t_j time units). A job j that finishes before its deadline d_j receives profit p_j, while a tardy job receives no profit (and might as well not have been scheduled).

   Give an efficient algorithm to find a schedule that maximizes total profit. You may assume that the t_j are integers in the range $[1, n]$. You should be able to come up with an $O(n^3)$ algorithm if you also assume that the latest deadline is $D = \sum_{1 \le k \le n} t_k$. This assumption is reasonable: (i) any job with deadline $> D$ can have its deadline changed to D without affecting its ability to be scheduled on time; and (ii) it takes $O(n)$ pre-processing to make any necessary changes in deadlines.

   In order to get full credit you need to come up with (a) a recursive formula for the maximum profit (based on an appropriate decomposition of the problem into smaller instances); (b) give details of an algorithm that fills the table; (c) show how your algorithm works on a small example; and (d) analyze the time bound.

   a. Let i represents the number of jobs and j represents the total time taken to execute the jobs. Create a 2D matrix P[0..i,0..j]. $P[i, j]$ represents the sum of profits of completing i jobs in j time units. p_i represents the profit incurred by that job if finished before the deadline. t_i represents the time taken by that job to execute and d_i represents the deadline for that job.

   There are four cases:
   1. If i = 0 or j = 0, i.e, first row and first column of P will be initialized to 0.
   2. If $t_i \le j \le d_i$, then we will select the maximum of $P[i-1, j]$ or $P[i-1, j-t_i] + p_i$.
   3. If $j < t_i$, i.e, if the job has not been scheduled yet, then the profit will be same as the above row.
   4.If the job has crossed the deadline, i.e, $j > d_i$, then we will set profit as P[i,j-1].

   The recursive formula for P[i,j] can be given by:

$$P[i,j] = \begin{cases} 0, & \text{if i = 0, j = 0} \\ \max(\text{P[i-1, j]}, \text{P[i-1, j-}t_i\text{]}+p_i), & \text{if } t_i \le j \le d_i \\ \text{P[i-1, j]}, & \text{if } j < t_i \\ \text{P[i, j-1]}, & \text{if } j > d_i \end{cases}$$

   b. Each job has a deadline, profit and the time required to execute them. The jobs are supposed to be finished within the deadline in order to earn the profit. Initially, we are sorting

the jobs in the order of their deadlines. Let us assume that there are i jobs which are supposed to be executed in j time units such that the profit is maximized. Initially,we will create a matrix P[0..i,0..j].

Now, there can be four cases to find the maximum profit obtained by performing i jobs in j time units.
Case 1: There are no jobs to execute or there is no time to execute,i.e, if i = 0 or j = 0. In such cases, the profit incurred will be 0.
Case 2: If the time to execute i jobs, i.e., j, lies between the time required by i^{th} job to execute(t_i) and the deadline associated with it(d_i), then we will select the maximum of the profit of the job above it(P[i-1,j]) and the sum of profits of executing $i-1$ jobs with $j-t_i$ time left($P[i-1, j-t_i]+p_i$). This recursive call will decompose into smaller instances which we can easily access via looking up the array.
Case 3: If the time to execute i jobs,i.e., j, is less than the time required by i^{th} job to execute(t_i), then we will select the profit incurred by the job above it, i.e., P[i-1,j]. In other words, if the job has not been scheduled yet, then we will select the profit of the above row.
Case 4: If the time to execute i jobs,i.e., j, is greater than the deadline associated with the i^{th} job(t_i), then we will select the profit incurred by the job to the left of it, i.e., P[i,j-1]. This will help us to keep in mind the previous profit incurred after the deadline has passed. This case is essential because if we have to schedule the jobs in which the total time given is more than the time actually required to finish the jobs, then we need to keep a track of the value on the left side.
So, in order to find P[i,j], i.e., profit earned by executing i jobs in j time units, we will keep on calling the recursive function until we find the base condition. Our maximum profit will be at the bottom right corner of the matrix, or the last element of the matrix.

To find the set of jobs which maximizes the profit, we need to backtrack from the last co-ordinate and see while traversing upwards till the first row. We will see the values which are getting affected while traversing and if there is a change in the transiting values, then that job is contributing towards the profit.

c. Let us assume that the jobs are sorted in the increasing order of the deadline.

Job id	Time Units	Deadline	Profit
1	2	2	3
2	3	5	4
3	1	6	5

Suppose we want to find the maximum profit in 4 time units and the set of jobs associated with it.
Create a matrix P[0..3,0..4] Here, i = 3 and j = 4. So, we have to find P[3,4].

0	0	0	0	0
0	0	3	3	3
0	0	3	4	4
0	5	5	8	9

P[0,0] = 0 (Since $i = 0$)
P[0,1] = 0
P[0,2] = 0
P[0,3] = 0
P[0,4] = 0
P[1,0] = P[0,0] = 0 (Since $j = 0$)
P[1,1] = P[0,1] = 0 (Since $j < t_i$)

P[1,2] = max(P[0,2],P[0,0]+3) = 3
P[1,3] = P[1,2] = 3 (Since $j > d_i$)
P[1,4] = P[1,3] = 3
P[2,0] = 0
P[2,1] = 0
P[2,2] = P[1,2] = 3
P[2,3] = max(P[1,3],P[1,0]+4) = 4
P[2,4] = max(P[1,4],P[1,1]+4) = 4
P[3,0] = 0
P[3,1] = max(P[2,1],P[2,0]+5) = 5
P[3,2] = max(P[2,2],P[2,1]+5) = 5
P[3,3] = max(P[2,3],P[2,2]+5) = 8
P[3,4] = max(P[2,4],P[2,3]+5) = 9

Hence, the maximum profit is 9.

Backtracking from P[3,4]. The trace will be P[3,4]→ P[2,3] → P[1,0] → P[0,0] and since P[0,0] = 0, we will stop. We can see that there is a change in values while transiting from job 3 to job 2 and job 2 to job 1, the jobs that contribute to the profit are Job 2 and 3. There is no transition of profit from P[1,0] to P[0,0]. So, job 1 doesn't contribute towards the profit.

Hence, executing job 2 and 3 when we have 4 time units will maximize the profit to 9.

d. The complexity of the algorithm will depend upon the filling up the table. For each job, we will fill the row till the deadline and since there are n jobs, the complexity comes out to be:

$$f(n) = n * \sum_{i=1}^{n} t_i$$

where $\sum_{i=1}^{n} t_i$ represents the width of the column and since we are given that t_i varies from [1...n], we can write

$$f(n) \leq n * \sum_{n=1}^{n} n = n^2 * n = n^3$$

Hence, by the definition of Big-Oh, $f(n) = \mathrm{O}(n^3)$, c = 1 and $n > n_0 = 0$

2. [5 points for the recursive formula; 4 points for the algorithm; 1 point each for the examples and the analysis]

Purpose: Applying dynamic programming to a new problem.

Suppose you have three strings $X = x_1 \cdots x_m$, $Y = y_1 \cdots y_n$, and $Z = z_1 \cdots z_{m+n}$. String Z is said to be a *shuffle* of X and Y if it can be formed by interspersing characters of X and Y while maintaining the original order of each string. For example `blmuondeasy` and `mbolnudeasy` are shuffles of `monday` and `blues`, but `blmuondeysa` is not (the `a` and `y` of `monday` are in the wrong order). Give an algorithm that determines, given inputs X, Y, and Z, whether or not Z is a shuffle of X and Y. Illustrate how your algorithm works on $X =$ `my`, $Y =$ `dog` and $Z =$ `domyg`; also do this with $Z =$ `dmogy`. What is the asymptotic runtime of your algorithm as a function of m and n? In order to get credit for your algorithm you will need to give a recursive formula that yields true or false for the problem based on results of appropriate smaller instances. Note: the letters in the two strings need not be distinct. For example, you could have $X =$ `love` and $Y =$ `dog`.

a. We are given three strings, X,Y and Z with length m,n and m+n respectively. We have to check if Z is a shuffle of X and Y if it can be formed by interspersing the characters of X and Y while maintaining the relative order of each string. We will create an array M[0..m,0..n] with all the values, except the first index, M[0][0], will be initialized as False. Now, we will keep on comparing the values of (X,Z) and (Y,Z) respectively.

1. Given three strings X, Y and Z of size m,n and m+n respectively.
2. Create M[0..m,0..n]. Initialize all the values of M as false except M[0][0]
3. For i from 0 to m:
3.1 For j from 0 to n:
3.1.1 If j = 0 OR (X[i-1]=Z[i+j-1] AND Y[j-1] ≠ Z[i+j-1])
3.1.1.1 Set M[i][j] ← M[i-1][j]
3.1.2 If i = 0 OR (X[i-1]≠ Z[i+j-1] AND Y[j-1]=Z[i+j-1])
3.1.2.1 Set M[i][j] ← M[i][j-1]
3.1.3 If X[i-1]=Z[i+j-1] AND Y[j-1]=Z[i+j-1]
3.1.3.1 Set M[i][j] ← (M[i-1][j] OR M[i][j-1])
4. Return M[m][n]

b. For X = my, Y = dog and Z = domyg
Creating an array M[0..2,0..3]
Initializing all the elements of M to False except M[0][0]
M[0][0] = True
For d: M[0][1] = M[0][0] = True
For o: M[0][2] = M[0][1] = True
For m: M[1][2] = M[0][2] = True
For y: M[2][2] = M[1][2] = True
For g: M[2][3] = M[2][2] = True
Result: M[2][3] = True

T	T	T	F
F	F	T	F
F	F	T	T

For X = my, Y = dog and Z = dmogy
Creating an array M[0..2,0..3]
Initializing all the elements of M to False except M[0][0]

M[0][0] = True
For d: M[0][1] = M[0][0] = True
For m: M[1][1] = M[0][1] = True
For o: M[1][2] = M[1][1] = True
For g: M[1][3] = M[1][2] = True
For y: M[2][3] = M[1][3] = True
Result: M[2][3] = True

T	T	F	F
F	T	T	T
F	F	F	T

c. This algorithm will consists of nested for loops of length of the strings X and Y, i.e., m and n respectively. After than, we will be using a if condition to check for the cases. Hence, the asymptotic run time of the dynamic approach comes out to be O(mn).

d. We are given three strings, X Y and Z of size m, n and m+n. Let M[0..m,0..n]. So, we are supposed to find the last element of the bottom-most corner,i.e., M[m][n]. We will start from the first index, i.e., m=0 and n=0 and assign it true. The rest of the matrix will be initialized to False. Now, we will keep on iterating over the values of X, Y and Z and we will consider various cases. The three cases include:

1. If the character of X matches with the character of Z and not with the character at Y $\rightarrow$ it will be equal to the element at the top of it
2. If the character of Y matches with the character of Z and not with the character at X $\rightarrow$ it will be equal to the element at the left of it
3. If the character of X and Y matches with the character of Z $\rightarrow$ it will be equal to the Logical OR of the element at the left and top of it.

Therefore, the recursive formula of this problem can be given by:

$$M[m,n] = \begin{cases} \text{True}, & \text{if m = 0 and n = 0} \\ \text{M[m-1,n]}, & \text{if X[m-1]=Z[m+n-1] or n=0} \\ \text{M[m,n-1]}, & \text{if Y[n-1]=Z[m+n-1] or m=0} \\ \text{M[m-1][n] } \| \text{ M[m][n-1]}, & \text{if X[m-1]=Z[m+n-1] and Y[n-1]=Z[m+n-1]} \\ \text{False}, & \text{Otherwise} \end{cases}$$

3. [3 points for part (a), 4 points for (b), 2 points for (c), 1 point each for (d) and (e)]

Purpose: Developing a greedy algorithm for a new problem.

Suppose you have a long straight country road with houses scattered at various points far away from each other. The residents all want cell phone service to reach their homes and we want to accomplish this by building as few cell phone towers as possible.

More formally, think of points $x_1, \ldots, x_n$, representing the houses, on the real line, and let d be the maximum distance from a cell phone tower that will still allow reasonable reception. The goal is to find points $y_1, \ldots, y_k$ so that, for each i, there is at least one j with $|x_i - y_j| \leq d$ and k is as small as possible.

Come up with a greedy algorithm for this problem. If the points are assumed to be sorted in increasing order your algorithm should run in time $O(n)$. Be sure to describe the greedy choice and how it reduces your problem to a smaller instance. Prove that your algorithm is correct.

In order to get full credit you need to (a) give a recursive formula that describes how the greedy choice reduces the problem to a smaller instance; (b) describe the greedy choice and prove the greedy choice property; (c) give details of the algorithm; (d) show how your algorithm works on a small example; and (e) analyze the time bound.

a. Let X be array of house location($x_1, \ldots, x_n$) and Y be the array of towers location. d is the maximum distance from a cell phone tower for a reasonable reception. We are required to find the minimum number of towers which need to be installed. For that, the first tower installation will take place at d distance from the first house. Any subsequent tower will be installed if the last tower's signals don't reach the current house. The installation of any such tower will take place at d distance from the current house. In this, we can see that the installation of any new tower depends upon the last installed tower. This gives us the greedy recursive formula for finding the tower location.

$$Y[j] = \begin{cases} \text{X[1]+d,} & \text{if j = 1} \\ \text{X[i]+d} & \text{if abs(x[i]-y[j-1])>d} \end{cases}$$

b. The greedy choice is the placement of the tower at the maximum distance from a house which has no signal reception. If we find a house which has no signal reception, then the next tower will be placed at d distance on the right, which is the maximum reception range. This will not only help us to fulfill the network reception requirement of the current house but also the houses till the d distance from tower on the right side. In short, using this greedy approach helped us to cover $2d$ distance which is the maximum distance. This placement of tower will help us to cover all the houses in the range of $2d$ from the current house and our problem is divided into further sub-problems where we are supposed to find the next tower placement after the $2d$ distance from the current house.
If we are not following the greedy approach, we will be installing tower at each house which doesn't receive the signal. This will increase the number of towers. Moreover, the maximum signal strength a tower can reach will be d rather than $2d$ distance in the previous case.

c. The installation of first tower will be at the maximum distance d from the first house on the right side. This will cover all the houses at a distance of $2d$ from the first house and will further reduce it into a sub-problem. Then for each subsequent house, we will check if the house has any signal receptor from the last tower installed. If there is, then no new tower for that house will be installed. For each house which doesn't have the signal reception on the last installed tower, we will install the tower at the maximum distance from that house within the range, i.e., d. This greedy approach will divide our problem into sub-problem and will help us to find the location of towers.

d. Let us assume that X = [2,12,15,29,54] and d = 10. Let Y be the location of towers installed.
Using the greedy approach, the first tower will be installed at 2+10 = 12,i.e., Y= [12].
Now, 12 and 15 has the range of tower at Y[0]. So, no tower will be installed for them. Here, the greedy approach has helped us to reduce the problem into a smaller subset.
House at 29 has no network reception from the tower at 12. Therefore, a tower will be installed at 29+10 = 39. So, Y = [12,39].
House at 54 again will not have any network reception from the tower at 39. So a tower will be installed at 54+10 = 64. This gives us, Y = [12,39,64]

e. Since the houses are sorted in increasing order, we have to use only one loop to check if the house has the range of tower or not. We are actually comparing the distance between the current house and the last installed tower. If the current house has no signal receptor, we will be installing a tower at d distance. This will continue till we reach the last house in the list. So, the run time complexity of this algorithm will by O(n) where n is the number of houses.