Anar Rzayev Introduction to Algorithms ID: 20180788

Homework 5 - CS300

1) a) We will explain our algorithm in subsequent pages, and will present in detailed form why it works correctly. However, we should firstly define the synthesized sequence of $S = (1, 3, 2, 1, 2, 4, 1)$ that maximizes the sum of its elements $\Rightarrow (1, 3 \cdot 2, 1, 2 \cdot 4, 1) = (1, 6, 1, 8, 1)$ where sum = 17 takes the reachable maximum among the other choices of synthesized sequences ✓

b) We'll construct our DP table as follows:
For each i, Dp[0][i] defines the synthesized sequence of the first i elements that maximizes the sum of its elements, but with a condition that i^{th} element will not be paired up, i.e. our synthesized sequence's last element should be S[i], and it's not paired with $(i-1)^{th}$ element (By paired, it's meant that they are not multiplied) to form desired sequence. Similarly, Dp[1][i] defines the synthesized sequence of the first i elements, which maximize the sum of its elements, but with a condition that

i^{th} element should be multiplied with $(i-1)^{th}$ element to form that sequence. In other words, in order to get the maximum sum of elements from the first i numbers' synthesized sequence, we defined Dp[0][i] and Dp[1][i], respectively, where former defines the sum with a condition that i^{th} element is not multiplied, and it's last element of the synthesized sequence that produce maximal sum Dp[0][i]. However, latter defines the sum with condition that i^{th} and $(i-1)^{th}$ elements are multiplied to form the desired synthesized sequence out of the first i elements, where resultant sum = Dp[1][i] and sequence's last element is S[i] * S[i-1]

It's crucial to note that if |S|=n, then our DP table will consist 2n entries, and each Dp[][j] entries will define the maximal possible sum of the first j numbers' synthesized sequence, whether the last element was multiplied by previous number, or not, will determine Dp[1][j] and Dp[0][j] values for each column of this DP table. Note that, values of j range from 1 to n (array indices start at 1 for S)

So, basically, $Dp[0][j]$ and $Dp[1][j]$ (for each $j \in [1,n]$) will determine columns from $j=1$ to n, according to whether last element was multiplied or not. It's crucial to observe that we only define maximal sums of synthesised sequences of the first j numbers for each entry, and in the last step, we'll show detailedly how we can construct synthesized sequence for the set S with $|S|=n$, given we have found all the required summations ✓

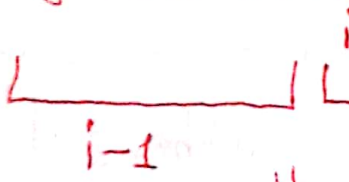
c) Initially, we have to define base cases for our DP. Since there are no elements for $Dp[\,][0]$ (0 elements are chosen) we can take $\boxed{Dp[0][0] = Dp[1][0] = 0}$ Moreover, if we just choose 1 element, $Dp[0][\,]$ and $Dp[1][\,]$ will not matter, since there are not any two elements to exclude them. Therefore, one element will contribute to the maximal sum and that'll be first element of $S=$ i.e. element $S[1]$

$\boxed{Dp[0][1] = Dp[1][1] = S[1]}$ Now, assume $i \geq 2$, and iterating from $i=2$ to n, we'll give explicit formulas and details on how and why they are constructed in this way:

$$Dp[0][i] = \max(Dp[0][i-1], Dp[1][i-1]) + S[i] \quad \star$$

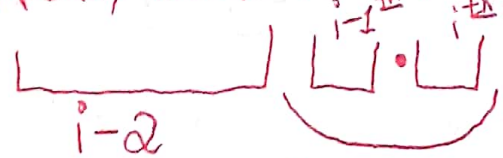
$$Dp[1][i] = \max(Dp[0][i-2], Dp[1][i-2]) + S[i-1]S[i]$$

where i ranges from 2 to n ($i=2,3,\dots,n$) Now, let's explain why this relationship holds



If i^{th} element was not multiplied with $(i-1)^{th}$, then it's left isolated as a single element for the synthesized sequence of the first i numbers, and therefore, for computing the possible maximal sum, we have to take the first $(i-1)$ numbers, compute the maximal possible sum for the synthesized sequence of these $(i-1)$ numbers, and add up that value to the i^{th} element. Since there are only 2 possible choices for determining the max sum for the synthesized sequence of first $(i-1)$ numbers (i.e. $Dp[0][i-1]$ and $Dp[1][i-1]$) $Dp[0][i]$ - possible max sum where i^{th} element was not multiplied - becomes adding up $\max(Dp[0][i-1], Dp[1][i-1])$ and $S[i]$, which is exactly what we wrote ✓

Now, when i^{th} element was multiplied with predecessor



we know that it's not further allowed to multiply that product with other elements. There, we have to isolate $S[i-1] \cdot S[i]$ as a part of synthesized sequence, and add up that value to the possible max sum of the synthesized sequence of the first $(i-2)$ numbers

Since possible max sum can be any of Dp[0][i-2] and Dp[1][i-2], we have to take the maximum of these numbers to obtain the desired max sum for the synthesized sequence of the first (i-2) numbers, which is exactly what we wrote in the formula ✓

(Note that, the second part was just calculating Dp[1][i] in terms of other relations, where last element was paired up among the first i elements)

Hence, our relationships are valid in the provided DP table, and it allows us to evaluate the maximal possible sum of the synthesized sequence of the first i elements

Notice that i ranges from 2 to n, where Dp[0][n] and Dp[1][n] will be maximal possible sums of the given n numbers' synthesized sequence, but with a condition that whether last element was paired up or not

Thus, if we take max(Dp[0][n], Dp[1][n]), we'll know that it's the desired answer, giving us possible maximum sum of the elements of given n numbers' synthesized sequence. (since there are only 2 cases for it)

max(Dp[0][n], Dp[1][n]) will be the answer for determining the possible max sum of elements from the given n numbers' synthesized sequence ✓

d) def max_sum(given tuple S with n # of elements):
construct empty DP table with size 2x(n+1)
Let Dp[0][0]=Dp[1][0]=0
Let Dp[0][1]=Dp[1][1]=S[1]
for each i=2 to n:
Dp[0][i]= max(Dp[0][i-1], Dp[1][i-1])+S[i]
Dp[1][i]= max(Dp[0][i-2], Dp[1][i-2])+S[i-1]S[i]
return max(Dp[0][n], Dp[1][n])

def required_sequence(found Dp table from previous, given tuple S function call):
construct an empty list "new_list"
Let i=n
while i>0:
if Dp[0][i] >= Dp[1][i]:
new_list.append(S[i])
i = i-1
else:
new_list.append(S[i]·S[i-1])
i=i-2
return reversed_list_of(new_list)

- As it's seen from first function, we return the max possible sum, where our Dp table will be constructed for that value. When we pass to second function, it's crucial to observe that we return the desired tuple for that maximum possible sum.

Indeed, creating an empty list, and going through "while" loop from i=n, we check which of the values Dp[0][i] and Dp[1][i] is greater. If Dp[0][i] ≥ Dp[1][i], then our synthesized sequence should contain the i^{th} element alone, since maximum sum of the resultant sequence for the first i numbers did contain or did not contain S[i] alone, it depended which of Dp[0][i] and Dp[1][i] is greater And in this case, Dp[0][i] is a bigger value, meaning that last element for the first i elements of S (i.e. S[i]) would be isolated in the sum and sequence for synthesized ⇒ Thus, we added S[i] to current list (and decremented by 1) When Dp[0][i] < Dp[1][i], then in order to get the desired synthesized sequence for the possible max sum, we have to choose the configuration which states that Dp[1][i] is the answer ⇒ (of the maximum of these 2 numbers)

In fact, this tells us i^{th} and $(i-1)^{th}$ elements were multiplied in order to get this sum, and for obtaining the desired sequence, we append S[i-1]·S[i] to the new_list and decrement i by 2 (since 2 elements were used)

Finally, we found the desired list, but it's not given in increased order of indices. If it's allowed, we can also reverse the list by making new array and put new values one-by-one

e) From the algorithm, it is observed that initializing takes constant time, for-loop goes from 2 to n, and returning max value is constant-time. (This's first function)
In the 2nd algorithm, we create a new list with initializing i to n. Going through while-loop takes at most linear time, since we decrement either by 1 or 2. At the final stage, we can just return the resultant list (order of elements will be shown in reverse order) or create a new object with n dimension, and put back values in (one-by-one) the correct order. Any of these algorithms, as described, will be run on linear time, where |S|=n. Thus,

running time of given problem = O(n) ☆ Y • ☺

2) a) It's important to mention that this problem mostly resembles "knapsack problem without repetition", where we define people as weight, electors as value, and items as states. Note that we have limited # of space and in this problem, we're winning by popularity of electors (we need sufficient electors by winning some states) It's crucial to notice that we have to find minimum number of electors by which we have to win.

State	People	Electors
1	p_1	e_1
...	...	...
i	p_i	e_i
...	...	...
n	p_n	e_n

Take $q_i = \left\lfloor \frac{p_i}{2} \right\rfloor + 1$ $_{i=1,2,\dots,n}$ to be minimum # of people we should care on state i

Moreover, consider threshold value where it represents minimal possible value of electors from whom we have to get votes $T = \left\lfloor \frac{e_1 + \dots + e_n}{2} \right\rfloor + 1$

Coming back to the construction of DP table, its dimension will be $(q_1 + q_2 + \dots + q_n) \times n$ and constructed by

$Dp(q, j)$ = maximum electors achievable using with capacity of q people and states $1, 2, \dots, j$

b) For the initialization, we set $Dp(q, 0) = Dp(0, j) = 0$

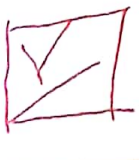
to be the base cases (which are clearly true due to the definition of DP)
Now, when we are evaluating $DP(q, j)$, it is clear that there will be either j^{th} state (item) or not. If j^{th} state (item) was counted, it's logical to express $DP(q, j)$ in terms of $DP(q - q_j, j-1) + e_j$ where q_j is min # of people on state j, and we used $(j-1)$ since we'll not consider state j again (no repetition)
Similarly, if j^{th} state (item) wasn't counted, then we just return $DP(q, j-1)$. The following is the relationship between $DP(q, j)$ and smaller entries:

$$DP(q, j) = \max\left(DP(q, j-1), DP(q - q_j, j-1) + e_j\right) \quad \text{for each } 1 \le j \le n \quad (*)$$

c) Our function will take inputs as the set of all people $p_1, p_2, \ldots, p_n$; the set of all electors $e_1, e_2, \ldots, e_n$; and # of items n. The next page will illustrate the code snippets for this algorithm, and will provide what is the running time of it:

```
def (n, (e1, e2, ..., en), (p1, p2, ..., pn)):
  let X=0, y=0,
  for i=1 to n:
     qi = floor(pi/2) + 1
     X = X + qi
     y = y + ei/2
  create a DP-table with dimension (X, n)
  let T = floor(y) + 1
  for each i=1 to n:
    Dp(0, i) = 0
  for each j=1 to X:
    Dp(j, 0) = 0
  for w=1 to X:
    for j=1 to n:
      if wj > w: Dp(w, j) = Dp(w, j-1)
      else: Dp(w, j) = max(Dp(w, j-1), Dp(w-wj, j-1) + ej)
      if Dp(w, j) >= T and j == n:
        return w          (## smallest # of people to obtain T electors)
```

Considering $Dp(X, n) = e_1 + e_2 + \dots + e_{n-1} + e_n$ is true, we have no doubt that return value will not be "None"



d) For each entry of DP-table, there exists O(1) time of work, and we go through each iteration of DP-table, thus running time is $O((q_1+q_2+\dots+q_n)\times n)$ (entry) ★ or just $O\left(\left(\left\lfloor\frac{p_1}{2}\right\rfloor+1+\dots+\left\lfloor\frac{p_n}{2}\right\rfloor+1\right)\times n\right)$, rewriting it as

running time → $O\left(\left(\sum_{k=1}^{n}\left\lfloor\frac{p_k}{2}\right\rfloor+1\right)\times n\right)$ ★ V ■ +