Intro to Algo HW #5

Sunday, April 5, 2020 2:51 PM

Problem 6.4

a)

We need to solve the sub problem of figuring out if a given amount of characters in the string array actually constitutes a word. We solve this by checking the previous index j that constitutes a real word, and seeing if $j \rightarrow i$ is a real word. And either returning $0 \rightarrow i$ or $0 \rightarrow j$ depending on which is larger.

The algorithm is as follows:

```
answer = [0, .... , n]
for i from 0 to n:
        // j<i
        answer[i] = max(valid - S[1 → j] , dict[j + 1 → i] = True)

return(answer[n-1])
```

What is happening here is that the index of the answer array is being updated at i as a valid location if the substring from j to i is valid in the dictionary. Because we piece together the array based on the substrings this is solving the sub problem.

Analysis: We run the loop for n elements, and we are solving n sub problems each time. This is a $O(n * n) \rightarrow O(n^2)$ algorithm

b)

The algorithm is as follows:

```
answer = [0, .... , n]
index = [0, ... , n]
for i from 0 to n:
        // j<i
        answer[i] = max(valid - S[1 → j] , dict[j + 1 → i] = True)

if answer[i]== True:
        index[i] == j
return(answer[n-1])

//printing
count = 0
for i =0 to n:
        if index[i] >0:
                print(S[count:index[i]])
        count = index[i] +1
```

What is happening here is that the algorithm is calculating if the string consists of valid words, with the addition of storing the location when a new valid word is found. We use those indices to print.

Analysis: The main algorithm is as shown above with another line that is $O(1)$ time, and printing is $O(n)$ which means runtime is still $O(n^2)$

Problem 6.8

The characteristic sub problem is to determine if the sub string from (i-1,j-1) in both X and Y are common.

The algorithm is as follows:

```
Define a table that will hold lengths and indices of common substrings.
for i=0 to n:
    for j=0 to m:
        if x[i] == y[j]:
            Table[i-1][j-1] = Table[i-1][j-1] +1
        else:
            Table[i-1][j-1]=0

return (max(Table), max(Table_index_i), max(Table_index_j)
```

The algorithm checks all possible combinations of indexes, and updates the table when a match is made. Otherwise if a match is broken then only at that specific table index will it be set to 0. This way when we return the max of the table in terms of i and j we are getting the indexes where the substring exists in x and in y and the substring itself.

Analysis:

We run the loop for $n * m$ times and solve the $n * m$ subproblems in $O(1)$ time. Thus the overall runtime is $O(n * m)$

Problem 6.13

a)

A sequence that is will result in Player 1 losing when taking the greedy approach is {2,3,1,4,5,9,7,8}

Running the greedy algorithm on this we find that player 1 will end up with 17 and player 2 ends up with 22. Thus the greedy algorithm is not optimal in all cases.

b)

The main sub problem is finding the largest possible difference between player 1 and player 2. That is, finding the total the first player can choose minus what the second player can choose. If player 1 is going to win that result will be positive.

The algorithm is as follows:

Define a 2D matrix that stores, at each of its slots, if player 2 is in the lead or if player 1 is in the lead, based off their moves

Then for all elements in S we put it into the matrix M at index [i][i] and make it equal to $S[i]$

Then two pointers are set i and j where i=0 and j = n

for i<=j:

M[i][j] = max(S[i] - M[i+1][j], S[j-1] - M[i][j-1])

return M[i][j]

The algorithm simulates taking the first card or the last card and compares it to the previously solved sub problem stored in the matrix. That sub problem is comparing to the different routes that could have been taken. That way at the end, we know we know the sequence of what to take for player 1 to win.

Analysis: The matrix is $O(n^2)$ and computing the sub problems takes $O(1)$ time. Because filling in the table is $O(n^2)$ the runtime is the same.

Problem 6.22

We take the Knapsack without repetition problem but instead of finding the maximum we return when we find a sequence that adds to our desired t.
The sub problem is then finding the sum of numbers chosen from A[1, ..., j]
All integers can be given the same weight so that we can use the same Knapsack algorithm

The algorithm is as follows:

```
Initialize the matrix to 0 at every (0,j) and at every (t,0)
Take in array of ints A
for j = 1 to n:
    for w=1 to t:
        if w[j] > w:
            Table[w][j] = Table[w][j-1]
        else:
            Table[w][j] = max(Table[w,j-1] , Table[w-w[j], j-1] +A[j])
        if Table[w][j] == t:
            return Table[w][j]
```

Like the knapsack problem this is computing the possible combinations of integer values in O(n*t) time. Weights are all equal, it serves as a pointer that way everything can be checked if need be.
Because we store the results in table we can compute the next entry in $O(1)$ time.

Analysis: This is the knapsack without repetition problem so it will run in $O(n * t)$ time.