CSCI 3104 Algorithms Name: Jeremy M. Hein
Problem Set 10
Fall 2020, CU Boulder ID: 810952267

Instructions

- This problem set is **open book**: you may refer to the lectured material found on Canvas and the recommended books to help you answer the questions.
- This problem set is an **individual effort**. You must arrive at your answers independently and write them up in your own words. Your solutions should reflect your understanding of the content.
- **Posting questions to message boards or tutoring services including, but not limited to, Chegg, StackExchange, etc., is STRICTLY PROHIBITED. Doing so is a violation of the Honor Code.**
- Your solutions must be submitted typed in LaTeX, **handwritten work is not accepted**. If you want to include a diagram then we do accept photos or scans of hand-drawn diagrams included with an appropriate `\includegraphics` command. It is your responsibility to ensure that the photos you obtain are in a format that pdflatex understands, such as JPEG.
- The template tex file has carefully placed comments (% symbols) to help you find where to insert your answers. There is also a **STUDENT DATA** section in which you should input your name and ID, this will remove the warnings in the footer about commands which have not been edited. You may have to add additional packages to the preamble if you use advanced LaTeX constructs.
- You must CITE any outside sources you use, including websites and other people with whom you have collaborated. You do not need to cite a CA, TA, or course instructor.
- Take care with time, we do not usually accept problem sets submitted late.
- Take care to upload the correct pdf with the correct images inserted in the correct places (if applicable).
- Check your pdf before upload.
- **Check your pdf before upload.**
- **CHECK YOUR PDF BEFORE UPLOAD.**

Quicklinks: 1 2 3 4

Problem 1

Consider the problem of determining if one sequence is a subsequence of another. That is, for two sequences X and Y of length n and m respectively given to you as arrays, the problem is to determine if Y is a subsequence of X. The goal here is to demonstrate the *optimal substructure* property for this problem. To receive full credit you must

(a) Given an English description of some useful set of subproblems.

(b) Argue how solving all these subproblems will answer the original problem.

(c) Demonstrate using a recurrence relation (with bases cases) or a naïve recursive algorithm (with base cases) that the problem has the optimal substructure property.

(d) Briefly argue why your optimal substructure description is correct (justify why your recurrence or algorithm is correct and handles all cases).

A useful set of subproblems would be subsequences of X and Y in the form $X[0...i]$ and $Y[0...j]$ where $0 \leq i \leq n$ and $0 \leq j \leq m$. This is helpful in solving the original problem because if $Y[0...j]$ is a subsequence of $X[0...i]$ and $Y[j+1....m]$ is a subsequence of $X[i+1....n]$, then it must be the case that Y is a subsequence of X. Therefore, by solving all of the subproblems we will have solved the original problem.

Consider the following psuedocode:

```
naiveSubsequence(X, Y):
  let n = len(X)
  let m = len(Y)

  if m = 0:
     return TRUE
  else if m > n:
     return FALSE

  for i = 0 in n:
     if x[i] = y[0]:
         naiveSubsequence(X[i+1...n], Y[1...m])

  naiveSubsequence(X[n...n], Y[0...m])
```

Our base cases are when len(Y) = 0 and when len(Y) > len(X). Note that when len(Y) = 0, Y is empty and an empty subsequence is a subsequence of every array and therefore we must return TRUE. Alternatively, when len(Y) > len(X) we know that Y contains more elements than X, and therefore cannot be a subsequence of a smaller array, so we return FALSE.

We then loop through X starting at the beginning of the array and compare each element to the first element in Y. If we find a match, we recurse on two smaller arrays, understanding that Y[0] is a subsequence of X[0...i]. We then must compare the remaining parts of the arrays X[i+1...n] and Y[1...m] in the recursive call. Continuing this process is, in essence, solving subsequences in the form X[0...i], Y[0...j].

By dividing our subproblems in this manner, we either reach a point where we have exhausted all elements in Y, in which case Y is in fact a subsequence of X, or we exhausted all (or enough to know that X contains more elements than Y) elements in X, in which case Y is not a subsequence of X.

Note that in the case we finish examining elements of X and have not found a match, we must do one final recursive call to hit a base case that will return FALSE.

Problem 2

The subset sum problem asks, given an array A of positive integers and a positive integer w, whether there is a subsequence of A such that the sum of all elements in the subsequence is w.

For example, consider the input array $A = [4, 15, 8, 16, 23, 42]$. If $w = 31$ then the answer is "yes" since there is a subsequence $[15, 16]$ where the sum is equal to $15 + 16 = 31$. However, if $w = 13$ then the answer is "no" since no subsequence of A has sum equal 13.

Consider the following subproblem and optimal substructure description:

> Let $n = \text{len}(A)$. For all $0 \leq \ell \leq \text{len}(A)$ and $0 \leq q \leq w$, we define $SS(\ell, q)$ to be TRUE if there exists a subsequence of $A[1..\ell]$ such that the sum of all elements in the subsequence equal q and FALSE otherwise. We observe that $SS(n, w)$ is the answer to the original problem. We observe that our problem has the optimal substructure property since
>
> $$SS(\ell, q) = \begin{cases} TRUE & q = 0 \\ FALSE & \ell = 0 \text{ and } q \neq 0 \\ SS(\ell - 1, q) & \ell \neq 0 \text{ and } A[\ell] > q \\ SS(\ell - 1, q) \text{ or } SS(\ell - 1, q - A[\ell]) & \text{otherwise.} \end{cases}$$
>
> Indeed, we know that $SS(\ell, 0) = TRUE$ since the empty subsequence has sum equal to 0 and the empty subsequence is a subsequence of every array. Likewise, we know that $SS(0, q) = FALSE$ if $q \neq 0$ since the only subsequence of the empty array $A[1..0]$ is the empty subarray. If $A[\ell] > q$ then $SS(\ell, q) = SS(\ell-1, q)$ since any subsequence that uses $A[\ell]$ would have total sum larger than the desired sum. Finally, we can otherwise consider taking or not taking the integer $A[\ell]$ to be in our subsequence. If we decide to take it, then we only need to find a subsequence of the remaining prefix with sum $q - A[\ell]$.

To complete the above into a full dynamic programming algorithm description, you need to identify an appropriate memoization data structure and a good ordering of the subproblems. For full credit on this problem do the following:

1. Identify how you can index the subproblems described above.
2. Pick a memoization data structure and briefly justify your choice.

3. Describe an ordering of the subproblems such that you can fill in your memoization data structure using the recurrence given above and never referencing an unknown subproblem. Briefly justify your ordering.
4. Take all the details and combine them into a dynamic programming algorithm *expressed in pseudocode*that solves subset sum problem.

I will utilize a 2D matrix data structure, where column q represents a desired sum, and row l represents a subsequence of array A. By using this structure, we can map boolean values to every combination of $0 \leq q \leq w$ and subsequences l.

To demonstrate the order of subproblems and the indexing, consider the case when $A = [1, 2, 3, 4]$ and $w = 4$:

	0	1	2	3	4
0	T	F	F	F	F
1	T				
2	T				
3	T				
4	T				

Note we can fill in our base cases right away. That is, when $q = 0$, the entire column should be TRUE, as per the directions. Likewise, row 0 (which denotes the empty subsequence of A), should be FALSE for all values of $q \neq 0$.

Now we can fill in our table row-by-row. Cell (1,1), we have $q = 1$, $l = \{1\}$ and $A[l] = 1$. Neither of the base cases apply, and we note that $A[l]$ is not greater than q, so our boolean value is $[l-1][q]$ or $[l-1][q-A[l]]$. In this case, $[l-1][q-A[l]] = [0][0]$, which is TRUE, so we would choose true for this cell. Likewise for cell (1,2), we have $q = 2$, $l = \{1\}$ and $A[l] = 1$. We note that $A[l]$ is not greater than q, so our boolean value is $[l-1][q]$ or $[l-1][q-A[l]]$. In this case, both options evaluate to FALSE, so we we fill in our cell with FALSE.

	0	1	2	3	4
0	T	F	F	F	F
1	T	T	F		
2	T				
3	T				
4	T				

We continue this process until the entire matrix is filled in, and we can do this without ever referencing an unknown subproblem:

	0	1	2	3	4
0	T	F	F	F	F
1	T	T	F	F	F
2	T	T	T	T	F
3	T	T	T	T	T
4	T	T	T	T	T

```
subsetSum(A, w):
  let D = a 2D array with dimensions of (len(A)+1) x (w + 1))

  for i = 1 in w + 1:
     D[0,i] = FALSE             #set row 0 to FALSE for
                                 for all values q not = 0
  for i = 0 in len(A) + 1:
     D[i,0] = TRUE              #set column 0 to TRUE for
                                 all subsets of A
  for l = 1 in len(A) + 1:
     for q = 1 in w + 1:
        if A[l] > q:
           D[l,q] = D[l-1,q]
        else:
           D[l,q] = D[l-1,q] || D[l-1, q-A[l]]

   return D[len(A)+1, w]
```

The procedure follows the logic discussed above, by setting our base cases (row 0 and column 0) first, and then looping through the array, left to right, to solve the subproblems in the appropriate order. We return the boolean value in the "bottom right" corner of the matrix, as that is the value that considers all elements in array A, and is the desired sum w.

Problem 3

Given an array of nonnegative integers A we say that a subsequence B of A is *doubling* if B is empty, a single element array or for all $1 \leq i \leq \text{len}(B) - 1$, $2 \cdot B[i] \leq B[i+1]$. Consider the following dynamic programming algorithm that solves the problem of finding the longest doubling subsequence in an array A.

> We define $DS(i, w)$ to be the length of the longest doubling subsequence in $A[i..\text{len}(A)]$ such that the first integer in the subsequence is at least twice as big as w. We observe that to solve the original problem, we want to compute $DS(1, 0)$.
>
> It follows that $DS(\text{len}(A)+1, w) = 0$ since the only doubling subsequence of the empty array is the empty array regardless of the value of w. If $1 \leq i \leq \text{len}(A)$ but $A[i] < 2 \cdot w$, then $DS(i, w)$ is equal to $DS(i+1, w)$ since $A[i]$ can't be in the optimal doubling subsequence sequence. Otherwise, $DS(i, w) = \max\{DS(i+1, w), DS(i+1, A[i])+1\}$ since the optimal doubling subsequence sequence could or could not use $A[i]$. We can then write
>
> $$DS(i,w) = \begin{cases} 0 & i = \text{len}(A)+1 \\ DS(i+1, w) & i \leq \text{len}(A) \text{ and } A[i] < 2w \\ \max\{DS(i+1,w), DS(i+1, A[i]) + 1\} & \text{otherwise.} \end{cases}$$
>
> Let W be the largest integer in A. We observe that each subproblem can be indexed by $1 \leq i \leq \text{len}(A)+1$ and $0 \leq w \leq W$. We will use a 2D array D of dimension $(\text{len}(A)+1) \times (W+1)$ as our memoization data structure. We can set $D[\text{len}(A)+1, w] = 0$ for all $0 \leq q \leq W$. Then our recurrence relation above shows that to compute $DS(i, w)$ we may need to know $DS(i+1, w')$ for some $w' \geq w$ when $1 \leq i \leq \text{len}(A)$. Hence, we can compute the solution to subproblems indexed by larger values of i before smaller values of i and we can compute the solution to subproblems indexed by larger w values before smaller w values.

Use the above description to write a dynamic programming algorithm *as pseudocode* that not only finds the length of longest doubling subsequence of A *but also returns the subsequence as an array.* Your algorithm should use $O(\text{len}(A)W)$ space and run in $O(\text{len}(A)W)$ time.

Hint: you could consider adding a data structure to store information which helps you backtrack and recover the subsequence itself.

```
1   doublingSS(A):
2     let W = the largest integer in array A
3     let D be a 2D array with dimensions ((len(A) + 1) x (W + 1))
4     let B be an array with size at most len(A)
5     let c = 0      #counter to track the current longest doubling ss
6
7     for i = 0 in W:
8        D[len(A) + 1, i] = 0     #set base cases to 0
9
10    for i = len(A) to 1:         #loop from len(A) down to 1
11       for j = W to 0:           #loop from W down to 0
12          if A[i] < 2j:
13             D[i,j] = D[i+1,j]
14          else:
15             D[i,j] = max(D[i+1,j], D[i+1, A[i]]+1)
16          if D[i,j] > c:
17             c = D[i,j]
18             B[i] = A[i]        #if the longest ss grows,
19                                we can add A[i] to B
20     return B
```

The pseudocode follows the implementation described above, solving subproblems indexed by larger values of i and w before smaller values of i and w, thus arriving at the original problem of $DS(1,0)$.

I chose to use a simple array B to store elements of A that are included in the largest doubling subsequence. Array D tracks the size of the largest current doubling subsequence, so whenever that number grows (tracked by variable c in the pseudocode), we can add $A[i]$ to the list of elements.

In terms of the complexity requirements, note that our highest order term will result from the nested for loop in lines 10 and 11 of the pseudocode. This loop will run len(A) x W times, which would result in $O(\text{len}(A)W)$ time, so our time complexity requirements are met. If we assume allocating a single element in an array takes constant time, then allocation of the 2D array in line 3 would also take $O(\text{len}(A)W)$ time. Likewise, the greatest memory requirement will come from the 2D array we use to store our boolean values, which is $O(\text{len}(A)W)$ as well.

Problem 4

When studying dynamic programming we often think of the strategy as an improvement over naïve recursion, but that still means we're thinking of recursion trees. Fortunately, we are giving you another opportunity to master **Standard 7** about using recursion trees to solve recurrence relations!

Note that the form of the recurrence relations that we ask you to solve for Standard 7 is usually a little different than the form we see in dynamic programming solutions: our examples for simple dynamic programming problems have recursive calls on inputs like $n-1$ and $n-2$ but below we have $n/4$ instead.

Use the recursion tree method to solve the following recurrence relation.

$$T(n) = \begin{cases} 5T(n/4) + n & n \geq 2 \\ 1 & n < 2. \end{cases}$$

By "solve" we mean compute an expression for $T(n)$ and write your answer in the form $T(n) = \Theta(f(n))$ for some closed-form function f.

Show all your calculations. You do not need to worry about rounding errors in your analysis. You do not need to give the details of any calculus tools that you use to justify asymptotic notation: it is fine to state e.g. "$2n^3 + n^{11/5} = \Theta(n^3)$ *because* $3 > 11/5$ *so* n^3 *is the higher-order term."*