Problem Set 4
data structures and algorithms 2 - fall 2023

cfr5spw
due november 3, 2023 at 11:59 pm

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: tmn7vs
Sources:
https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/
https://www.geeksforgeeks.org/dynamic-programming/
lecture slides, class, and office Hours
for q2 and 3:
https://www.youtube.com/watch?v=aPQY__2H3tE&ab_channel=Reducible

problem 1 *Dynamic Programming*

1. If a problem can be defined recursively but its subproblems do not overlap and are not repeated, then is dynamic programming a good design strategy for this problem? If not, is there another design strategy that might be better?

   Solution:

   —- ANSWER P1) part 1 —-
   Because Dynamic programming is better suited when it can avoid redundant calculations by reusing previously computed results a problem with subproblems that do not overlap will not be best solved using dynamic programming.

   Instead we should use a more simple recursive approach such as a divide and conquer approach that we have looked at previously in class. This works better since in divide and conquer each sub problem is solved independently and then later combined (as the name suggests) to get a solution for the main problem.

2. As part of our process for creating a dynamic programming solution, we searched for a good order for solving the subproblems. Briefly (and intuitively) describe the difference between a top-down and bottom-up approach.

   Solution:

   —- ANSWER P1) part 2 —-
   TOP-DOWN: In this approach we start with a bigger/main problem and break it down into smaller subproblems - then as we solve the smaller problems we store the solutions using memoization to avoid the tedious/redundant work. Any problem thats being repeated (from the subproblems) can be stored to avoid computation similar to what I explained above in part1 about dynamic programming. It says "(and intuitively)" in the question so Im also gonna use an example: we can think about solving a rubiks cube the main goal obviously is to match all the colors on each side together so we will break down the problem by solving first one side, then we will solve the first layer of each side touching the complete side, since were solving the same layer for all other sides we can simply learn the algorithm to solve one side and then apply the solution to the rest. (This isnt really how you solve a rubiks

cube but its the first thing that came to mind).

BOTTOM-UP: using the bottom up approach on the other hand involves starting with the smallest subproblems and solve them first - you then use the solutions to solve the overarching bigger/main problem - an example of this would be constructing the DNA base sequence from class on Halloween - only using the matching pairs table with the big-that (m*n) time comp.
– END OF P1 –

PROBLEM 2 *Birthday Prank*

Prof Hott's brother-in-law loves pranks, and in the past he's played the nested-present-boxes prank. I want to repeat this prank on his birthday this year by putting his tiny gift in a bunch of progressively larger boxes, so that when he opens the large box there's a smaller box inside, which contains a smaller box, etc., until he's finally gotten to the tiny gift inside. The problem is that I have a set of n boxes after our recent move and I need to find the best way to nest them inside of each other. Write a **dynamic programming** algorithm which, given a $fits(b_i, b_j)$ function that determines if box b_i fits inside box b_j, returns the maximum number of boxes I can nest (i.e. gives the count of the maximum number of boxes my brother-in-law must open).

Solution:

in order to solve this problem im gonna note that we cant really sort because we have no information about size or shape of the boxes so were going to use a top down approach that has a recursive call to a helper func 'maxBoxes' that it will call and return the final value of the maximum ammount of boxes from a chain of fitting boxes. We will also be using a memoization array 'scores' in order to store the value of already computed box fits to avoid redundant calulations.

```
func findMaxNestingBoxes(boxes):
n = length(boxes)
scores = [] of size n; with all values initialized to -1
maxNestedBoxValue = 0
for i=0 i to n-1:
maxNestedBoxValue = max(maxNestedBoxValue, maxBoxes(boxes, scores, i))
```

Our maxboxes func is as follows:

```
func maxboxes(boxes, scores, i):
if scores[i] != -1; return scores[i]; – this will ensure we are not doing that redundant calculation.

maxBoxValue = 1 – minimum number of boxes we can nest
for j from 0 to n-1:
if i does not = j and fits(bi,bj) is true:
maxBoxValue = max(maxBoxValue, 1 + maxBoxes(boxes, scores, j))
end of for loop.
scores[i] = maxBoxValue – will store the maximum chaing of box fits(i,j) that return true.
return maxBoxValue
```

note to grader: if we run this in python we need to tell maxBoxes what n is and also maxBoxes will go before our findMaxNestingBoxes function – with those fixes this seems like it works just fine. Although maybe I overcomplicated it a bit, I was told we could not sort.
.
.

.
–END OF P2–

PROBLEM 3 *Arithmetic Optimization*

You are given an arithmetic expression containing n integers and the only operations are additions ($+$) and subtractions ($-$). There are no parenthesis in the expression. For example, the expression might be: $1+2-3-4-5+6$.

You can change the value of the expression by choosing the best order of operations:

$$((((1+2)-3)-4)-5)+6 = -3$$
$$(((1+2)-3)-4)-(5+6) = -15$$
$$((1+2)-((3-4)-5))+6 = 15$$

Give a **dynamic programming** algorithm that computes the maximum possible value of the expression. You may assume that the input consists of two arrays: `nums` which is the list of n integers and `ops` which is the list of operations (each entry in `ops` is either `'+'` or `'-'`), where `ops[0]` is the operation between `nums[0]` and `nums[1]`. *Hint: consider a similar strategy to our algorithm for matrix chaining.*

Solution:

let begin by observing that in order to get the best solution we need to find the maximum value of all the possible out comes for this expression. Using a dictionary, we can store the values of the local maxiums and minimums for sub problems in the expression - this is going to help with our top-down approach and avoid redundancy.

— our function will be called maxVal and were going to declare 2 helper functions, calc to calculate the individual operations. And Best which will be our recursive function that will breakdown our expression into local minimums and maxiums that will be store in our scores dictionary memory and called upon repeated calculations
def maxVal(nums, ops):
scores =
finalMax, uselessvar = best(0, len(nums) - 1, scores)
— note the useless var is where we would store the localMinimum that we will calculate later, this is not necessary however since we just want our local max - however the localMin is still necessary for grabbing minmum values since if we have a '-' operator we are going to want to subtract a max from a min to get the largest possible value.
return finalMax

def calc(int1,int2,op)
if op == '+': return int1 + int2
else return int1 - int2

def best(low,hi,scores):
— init local Max and min to 0
localMax =0, localMin = 0
if our low == high we want to return nums[low or hi] (since this will just be the final value; only one value left in part)
— lets check our scores dict for our hi/lo values if they are present we will return them to avoid further computation.
if (low,hi) in scores:
return scores[(lo,hi)]

for i in range(lo, hi):
leftMax, leftMin = best(lo, i, scores)
rightMax, rightMin = best(i + 1, hi, scores)
— compute and store all combinations lmaxrmax,lminrmax,lmax,rmin,lminrmin our subparts
results = [
calc(leftMax, rightMax, ops[i]),
calc(leftMax, rightMin, ops[i]),
calc(leftMin, rightMax, ops[i]),
calc(leftMin, rightMin, ops[i])]
— now lets update the result for our sub-expression
localMax = max(localMax, *results)—use for spread operator ...from js to grab remaining args.
localMin = min(localMin, *results)
scores[(lo, hi)] = (localMax, localMin)
return localMax, localMin

similar to matrix chaining in order to find best(1,6) we need to break down the problem recursivley and store values of subproblems in our memoization table. This solution uses 'scores' in order to keep track of these results.

PROBLEM 4 *Stranger Things*

The town of Hawkins, Indiana is being overrun by interdimensional beings called Demogorgons. The Hawkins lab has developed a Demogorgon Defense Device (DDD) to help protect the town. The DDD continuously monitors the inter-dimensional ether to perfectly predict all future Demogorgon invasions.

The DDD allows Hawkins to predict that i days from now a_i Demogorgons will attack. The DDD has a laser gun that is able to eliminate Demogorgons, but the device takes a lot of time to charge. In general, charging the laser for j days will allow it to eliminate d_j Demogorgons.

Example: Suppose $(a_1, a_2, a_3, a_4) = (1, 10, 10, 1)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The best solution is to fire the laser at times 3, 4 in order to eliminate 5 Demogorgons.

1. Construct an instance of the problem on which the following "greedy" algorithm returns the wrong answer:

   $\texttt{BADLASER}((a_1, a_2, a_3, \ldots, a_n), (d_1, d_2, d_3, , \ldots, d_n))$:
   Compute the smallest j such that $d_j \geq a_n$, Set $j = n$ if no such j exists
   Shoot the laser at time n
   if $n > j$ then $\texttt{BADLASER}((a_1, \ldots, a_{n-j}), (d1, \ldots, d_{n-j}))$

   Intuitively, the algorithm figures out how many days (j) are needed to kill all the Demogorgons in the last time slot. It shoots during that last time slot, and then accounts for the j days required to recharge for that last slot, and recursively considers the best solution for the smaller problem of size $n - j$.

   Solution:

   P4 PART 1 —-
   if we use arrays, a = (3, 10, 4) and d = (8, 1, 12).

   using badlaser for the largest a sub i — i.e. 10, the smallest j such that dj $\geq$ an.

d3 = 12 is the smallest value that is $\geq$ to a2 = 10. so lets fire the laser at time t = 2 to eliminate 10 demogorgs. The laser will take 12 days to recharge so the subsequent attacks on days 1 and 3 will nott be addressed.

a better solution is to fire the laser on day 3 to eliminate 4 demogorgs with 12 days recharge time. This allows the device to be ready again for the attack on day 1. This solution manages to eliminate a total of 7 demogorgs, which is better than the 4 eliminated by the badlaser strategy.

P4 PART 1 —-END

2. Given an array holding a_i and d_j, devise a dynamic programming algorithm that eliminates the maximum number of Demogorgons. Analyze the running time of your solution. *Hint: it is always optimal to fire during the last time slot.*

Solution:

P4 PART 2 —-

Let DP[i] be the maximum number of demogorgs eliminated up to day i.

init: DP[0] = 0

recurrance: DP[i] = max(DP[i-1], ai + DP[i-di])
we can see that DP[i-1]considers not firing on day i, and ai + DP[i-di] considers firing on day i.

To get our solution we can iterate over all i, and compute DP[i] using the above recurrence relation. The value DP[n] will give the max demogorgs eliminated.

The time complexity should be O(n), where n is the number of days. This is because for each day i, we perform a constant, n work to compute DP[i].

P4 PART 2 —-END