COMP160: Algorithms, **Homework 7**

Contents

- This homework is a work of love designed by TAs of previous semesters (mainly lead by Mert Erden and Long Tran). We think it is an incredibly well designed exercise that practices all aspects of dynamic programming. It may seem longer than usual, but notice that you need only need to do 3 exercises (other ones are optional).
- Some exercises ask you to fill in a table of subproblems. In order to answer the questions you may have to add additional parameters to the table. It is perfectly fine to do so, but make sure to (*i*) clarify what these parameters represent and (*ii*) say how to extract the original table from the table with additional parameters.
- To obtain full credit, you must justify your answers. When describing an algorithm, do not forget to **state any assumptions that you make**, analyze its running time and explain why the algorithm is correct.
- Although not specifically stated, you can assume that we look for algorithms that are as fast as possible, and bounds as tight as possible.
- You may discuss these problems with others, but remember to write the answers on your own. In case of doubt, cite any source you used to do the assignment.
- Remember to submit each question in a separate page.

1 The Memo (Optional coding assignment)

In order to enter Algorithmia, you need to know how to build a memo. Building memos for dynamic programming problems is case specific. The main objective of this problem is to learn the pattern associated with how we can use them in the context of coding.

For this problem we've provided you with 3 examples of memoized functions (found in `examples.py` or `examples.cpp`) that you have seen in class: Fibonacci, LCS, and rod cutting respectively. We have also provided non-memoized versions of two functions alongside their recurrences. Your objective is to memoize these functions.

a. You're planning to rob houses along one side of a street, with house i containing v_i valuables. If you rob a certain house i, its security system will alert the 2 adjacent houses (house $i-1$ and house $i+1$), so you can't rob adjacent houses. Given a list V of n houses, how can you make off with as much loot as possible? Fortunately you took algorithms in college and learned DP. Define the subproblem $R(V, i)$ = the maximum amount of money you can rob from houses h_1 through h_i. The base cases and recurrence relation for R are:

$$R(V,i) = \begin{cases} 0 & \text{if i = 0, since there are no houses to rob} \\ V[1] & \text{If i = 1, since you're at the 1st house, you can only rob it} \\ max\{R(V,i-1), V[i] + R(V,i-2)\} & \text{Else rob from the current house or skip it} \end{cases}$$

Note: in class we have presented arrays as 1-indexed and thus for consistency in this exercise we did the same. When you implement this make sure to adapt it for 0-indexed arrays (as is the standard for c++/python). The autograder will mark it incorrect otherwise.

b. Given an array of positive integers A and a target value, return whether there is a subset where the numbers add up to the target value. The subproblem $SBS(A, t, i)$ = true when there is a subset of $A[1, i]$ which adds up to the target value t.

Suppose `A = [5, 3, 1, 10, 16]`, i is 3, and current target t is 13. Then $SBS(A, 13, 3)$ would use case 4 below, making two recursive calls: $SBS(A, 13, 2)$ and $SBS(A, 13 - A[3], 2)$.

$$SBS(A,t,i) = \begin{cases} True & \text{If } t = 0 \\ False & \text{If } i = 0 \text{ and } t \neq 0 \\ SBS(A,t,i-1) & \text{If } A[i] > t \\ SBS(A,t,i-1) \text{ or } SBS(A,t-A[i],i-1) & \end{cases}$$

Note: For a theory discussion, it is fine for A to be implied/understood and not written as an explicit parameter of SBS since it does not change. This is how we wrote it in the lecture slides. For coding purposes however SBS will need to take A as input, so we have written it that way here. It is correct either way.

We've provided non-memoized versions of the problems above, but the autograder will time out, because the recurrence repeats too many cases. Your job is to write `memo_rob and memo_SBS` found in `memo.py or memo.cpp` which will record function calls with a memoizer so we can avoid repeated calls.

2 Magical Stones

Our story begins with our adventurer seeking to collect magical stones. These magical stones each have a damage value associated with them and can harm many-headed dragons. Alas our adventurer is not quite physically *talented*, and as such needs to collect the fewest number of stones such that they can still slay the dragon.
The ancient algorithmic scriptures say that:

The Resting Dragon has exactly n health and only the wielder of the magical stones may claim its treasure.

Formally, there is a dragon with exactly n health. Your task is to find the minimum number of magical stones needed to inflict **exactly** n damage, as to not tire our adventurer. Stones are each associated with a fixed amount of damage it inflicts, and you are able to select a stone as many times as you want. The stone damages are given to you in an array. Keep in mind, we are asking for the minimum *number* of stones required to kill the dragon, **not** an exact list of stones. You can assume there always exists a set of stones that can kill the dragon.

a. Give the minimum number of stones needed for the following examples:

   (i) `stone_damages = [1, 2, 3, 4, 5], dragon_health = 4`

   (ii) `stone_damages = [1, 2, 4, 6, 17, 24], dragon_health = 203`

   (iii) `stone_damages = [4,5,7], dragon_health = 36`

b. Now we're going to try solving the Magical Stone problem recursively. We're going to try writing the function $MS(c, n)$ where c is the array of stone damages and n is the health we want to match.

   (i) What are our base cases (when can we stop recursing)?

   (ii) When do we have to recurse, and which parameter(s) are changing in each recursive call?

   (iii) Based on your answers in b(i) and b(ii), formulate a recurrence for this problem.

c. Run your recurrence on example a(iii) and draw out part of the function call graph (recursion tree). Draw all recursions for levels 0,1,2, then draw the subtree with root $MS(c, 15)$. Notice where there are repeated calls.

d. Suppose `stone_damages = [17, 4, 3]`, `dragon_health = 71237`. You find a tablet from a previous fallen adventurer, which contains the minimum number of stones needed for dragons with particular healths.

You notice that most of the tablet has been damaged, but you see the visible part towards the end:

`dragon_health`	71208	71209	71210	71211	71212	71213	71214	71215	71216	71217
`minimum_stones`	4191	4192	4192	4192	4192	4189	4193	4193	4190	4190

`dragon_health`	71218	71219	71220	71221	71222	71223	71224	71225	71226	71227
`minimum_stones`	4194	4191	4191	4191	4192	4192	4192	4192	4193	

`dragon_health`	71228	71229	71230	71231	71232	71233	71234	71235	71236	71237
`minimum_stones`										

Fill out the empty cells in the tablet using the information from the part that you can see. Explain (in 2-3 sentences) how you deduced those values.

3 Health Rituals


VERY adventurer needs a way to heal themselves, especially if the fight is going to take a long period of time. Fortunately, we stumble across a tablet containing a list of magical healing spells. Each magic spell will cost you an associated amount of energy to cast. Once a spell is cast it cannot be cast again.

We are dealing with two quantities, energy denoted as E and health as H. These are separate quantities. Casting a spell lowers our energy, and increases our health.

We want to maximize the increase in health that we can get from casting some combination of spells, while making sure we do not run out of energy. More specifically, there are n spells, with respective costs of $[C_1, C_2, C_3, ..., C_n]$ stored in array C and corresponding healing amounts of $[H_1, H_2, H_3, ..., H_n]$ stored in array ΔH.

a. (i) Find the most health we can gain with the following example:
`E = 80, C = [15, 30, 45, 65]`, `ΔH = [20, 25, 85, 110]`

(ii) Provide an example where we cannot greedily pick a spell. In other words, provide an example where if someone simply picks the spells that have the highest $\frac{\Delta H[i]}{C[i]}$ ratios, they will not maximize their healing.

b. Given arrays C and ΔH, we now want to compute the function $HR(E)$ that takes in our total energy E and returns the maximum health we can gain. Formulate a recurrence for such a function (or possibly a related function), clearly specifying the base case(s) and recursive case(s).

c. Use bottom-up DP to fill in a table and solve for the following example.
`E = 9, C = [7,2,9,3,1], ΔH = [3,2,3,1,2]`
Hint: This has some similarity to an example seen in lecture, though not the same.

d. What is the time complexity of finding the maximum amount of health we can get from casting spells?

4 Book of Spells (Optional coding assignment)

For unknown reasons, you realize that you're carrying a book of what seems to be spells in your backpack. Each spell, unfortunately, is written as a sequence of numbers (such as "1", "123", "345") and needs to be converted into human readable form.

In this coding section you will implement a function that returns the number of possible string interpretations, given a string of digits that represents a mapping from numbers to characters. Together with this document you should have the 4 files: `main.cpp`, `interpret.cpp` (and the counterparts `main.py`, `interpret.py` for python).

As a reminder, you will **ONLY** be submitting **EITHER** the file called `interpret.cpp` **OR** the file called `interpret.py` to Gradescope. If you submit anything else, the autograder **CANNOT AND WILL NOT** grade your submission. In addition, please **DO NOT** change the name or the function signature of the function we gave you.

Implementation details

In this question, we will use the following mapping from numbers to letters:

```
"1" -> 'a'
"2" -> 'b'
"3" -> 'c'
  ...
"26" -> 'z'
```

In addition, we will only consider lower-case letters for the question. Given this, here's an example of interpretSpell in action:

An example

```
input = "1234"
interpretSpell(input) = 3
```

The 3 ways to interpret the string `"1234"` are:

- `1 23 4`, which maps to the string `"awd"`
- `12 3 4`, which maps to the string `"lcd"`
- `1 2 3 4`, which maps to the string `"abcd"`

Notice that there are other ways to split this input, such as `123 4` or `12 34`, but these don't have corresponding mappings to the alphabet, so they're not legal.

What you need to do

You will implement either:

```
def interpretSpell(inputSpell : str) -> int:
    # TODO
```

or

```
unsigned long interpretSpell(string inputSpell) {
    // TODO
}
```

depending on whichever language you choose.

NOTE: we realize that brute forcing is an option, but it will time out the autograder. Simply put, you have implement a correct **AND** efficient algorithm.

Input constraints

Here are the constraints on the function `interpretSpell` and the input `inputSpell`:

- `inputSpell` will **NOT** be the empty string (it is worth considering how you would deal with this case though, as it would be a base case of your recurrence).
- `inputSpell` should return zero if the string cannot be converted. For example, the string `"0121231"` cannot be converted (the initial '0' cannot be interpreted) and thus should return 0.
- we expect `interpretSpell` to terminate (with the correct answer) in less than 1 second for `inputSpell` with length 200.

5 The Secret Code

ou finally approach the dragons lair, but notice that it has been sealed off. Upon closer inspection you notice that there is a string of length n. You remember a line from the ancient algorithmic scriptures:

To open the dragon's lair one must find the longest collection of characters that read the same back and forth.

For example, the longest collection (subsequence) of such characters in the string "$CEEPCE$" is "$CEEC$".

Describe a recurrence, and outline an algorithm to find the longest palindromic sub-sequence in a given string. Briefly discuss the runtime of your algorithm. Remember to give a well-structured answer!

Ensure your algorithm works for the strings "$PADPA$", "$IRSI$", "$ILIKETOMATOSOUP$".

6 The Dragon (Optional challenging problem)

ALAS, the dreaded n headed dragon of Algorithmia approaches you. The final echoes of the scriptures say:

The dragon will only die if it takes the most damage possible.

Upon further inspection you notice that each head has a damage value d_i. Killing the i^{th} head damages the dragon by $d_\alpha \cdot d_i \cdot d_\beta$, where α and β are the adjacent *remaining* heads. The 1^{st} head and the n^{th} head will only have 1 adjacent head since they're at the ends.

For example, if the dragon had damage values `d = [40, 15, 105, 170]`, the optimal solution would kill the heads in the following order:

1. We kill the 2^{nd} head ($d_2 = 15$), and that deals $40 \cdot 15 \cdot 105 = 63,000$ damage.
2. Now we're left with 3 heads: [40, 105, 170]. We kill the 2^{nd} head in this list ($d_2 = 105$) and that deals $40 \cdot 105 \cdot 170 = 714,000$ damage.
3. Now we're left with 2 heads: [40, 170]. We kill the 1^{st} head and that deals $40 \cdot 170 = 6,800$ damage.
4. Now we only have 1 head left: [170], so we kill the last head, dealing 170 damage.

In total, we dealt $63,000 + 714,000 + 6,800 + 170 = 783,970$ damage, which is the best we can do (you can verify this by hand).

Provide a recurrence and describe a **dynamic programming** algorithm to return the maximum damage you can deal to the dragon, given an array of damage values of the dragon's heads. Briefly describe the runtime of your algorithm. Remember to give a well-structured answer! You will still get some bonus points even if the answer is not correct.

Note: This is a significantly harder problem. Although the conceptual idea is the same as all previous exercises, finding a recurrence that allows for easy solution is much more challenging than before. Try to solve it once you have solved other questions. In an exam, we wold not pose a question of this level of difficulty unless it comes with additional hints and/or a way to make the problem easier.
Note 2: The main purpose of this exercise is to practice finding difficult recurrences. For this exercise, questions that do not use a dynamic programming approach will be awarded zero points (even if correct).
Hint: Make a recurrence formula that looks at the last head that is killed. See what subproblems it would generate.

Guide for optional coding assignments

How to submit

For **The Memo**, please submit **ONLY** the file `memo.cpp` or `memo.py` depending on your language of choice.

For **Book of Spells**, please submit **ONLY** the file `interpret.cpp` or `interpret.py` depending on your language of choice.

Take advantage of the autograder. You can submit an infinite number of times, so don't hesitate to try out potential solutions.

Because we have a lot of large tests, the autograder may take up to **3 minutes** to terminate. The autograder will still run even if you exit the page, so submit your work, close the tab and check back in 5 to 10 minutes for your result.

Expectations

In addition to correctness, we expect all of the functions submitted in both **The Memo** and **Book of Spells** to terminate within 1 second on the worst-case input of size 200.

Recommendations

We recommend taking a look at **The Memo** coding assignment before attempting **Book of Spells**. In the starter code for **The Memo**, we've provided several examples along with benchmarking tests.

When attempting **The Memo**, feel free to make edits to the benchmarking tests to test your own functions. With larger tests, don't hesitate to comment out the recursive solutions since these may take years to terminate.

When attempting **Book of Spells**, try to come up with a recurrence for the problem before coding. When coding, you may be able to adapt concepts based on examples from **The Memo**.

For C++ users

Please do **NOT** modify anything in the `.h` file (we will use the original one to compile). If you define any helper functions, write their function signatures at the top of your `.cpp` file.