Homework 4

Student: Haoyuan Du, Shijun Yang

Purpose:

- Practice writing recurrences (i.e., recursive solutions).
- Learn to design algorithms using dynamic programming techniques.
- Continue to develop your algorithm analysis skills.
- Solve more complex problems using dynamic programming.

General Homework Policies:

- This homework assignment is due by the deadline given in Canvas. Late assignments will not be accepted and will receive a 0.
- Submit two files through Canvas:
  1. The `pdf` for the written portion of the assignment (it should generated by modifying the the LaTeX file for this assignment).
  2. The completed `DynamicSum.java` and `PalindromicSequence.java` files. Before submitting please zip all the java files together and submit one `.zip` file. See the *Additional Programming Instructions* section below for more detailed instructions.
- You may choose to work with one partner on this assignment. If you choose to work with a partner, only one assignment should be submitted and you and your partner will both receive the same grade. Make sure to include your partner's name on the homework!
- If you choose to work with a partner, your assignment should be a true joint effort, equally created, and understood by both partners.
- You are not allowed to consult outside sources, other than the textbook, your notes, the Java API and the references linked from Canvas (i.e., no looking for answers on the internet).[1]
- Getting *ANY* solutions from the web, previous students etc. is *NOT* allowed.[1]
- You are not allowed to discuss this assignment with anyone except for your partner (if you have one) or the instructor.[1]
- Copying code from anywhere or anyone is not allowed (even previous code you have written). Allowing someone to copy your code is also considered cheating.[1]
- Your work will be graded on correctness and clarity. Write complete and precise answers and show all of your work.
- Questions marked (PRACTICE) will not be graded and do not need to be submitted. However it's highly recommended that you complete them.

[1] *See the section of the syllabus on academic dishonesty for more details.*

Homework 4

Student: Haoyuan Du, Shijun Yang

Homework Problems:

1. (5 points) St. Thomas Cafeteria Food Inc. is considering opening a series of restaurants along Summit Avenue. There are n viable locations along this road, and the positions of these locations relative to the start of the road are, in miles and in increasing order, given in an array $M = m_1, m_2, \ldots, m_n$. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i \in 1, 2, \ldots, n$. The profit for each restaurant is given in an array $P = p_1, p_2, \ldots, p_n$. The constraints are as follows:

   - At each location, St. Thomas Cafeteria Food Inc. may open at most one restaurant.
   - Any two restaurants must be at least k miles apart, where k is a positive integer. For example if $M = 1, 4, 6$ and k is 3 then you may open restaurants at locations 1 and 4 (and 1 and 6) but not at locations 4 and 6 together since these are not 3 miles apart.

   Give an efficient dynamic programming algorithm to compute the maximum expected total profit subject to the given constraints. Note that the inputs to your algorithm are the number of n of viable locations, the array of locations M, the array of profits, P, and the distant constraint k.

   (a) (1 point) Define your subproblems (the entries of the table) in words. For example, in the domino tiling problem the definition is $D(i)$ = the number of ways to tile a $2 \times i$ checkerboard.

   Solution: MaxProfit(i) is the maximum profits by opening the restaurants from M_1 to M_i, including M_i.

   (b) (2 points) State the recurrence for the entries of your table (NOT for the running time), and explain in words why it's true. For example, in the domino tiling problem the recurrence for the entries is $D(i) = D(i-1) + D(i-2)$.

   Solution: Firstly, we create an array L=L_0,L_1,...,L_n which stores the last possible restaurant's index in M (at least k miles apart) at L_i.
   Then, we create the MaxProfit array. If we open the i restaurant, the new profit MaxProfit(i) = $P[i] + MaxProfit[L[i]]$. If we don't open the i restaurant, the new profit MaxProfit(i) = MaxProfit(i-1). So, the best profit will be:
   MaxProfit(i) = max(MaxProfit(i-1), $P[i] + MaxProfit[L[i]]$).

   (c) (1 point) Give pseudocode for filling in the table AND returning the solution. Do NOT use recursion instead use iteration.

   Solution:

   function GetMaximumProfit($M[m_1, ..., m_n]$, $P[p_1, ..., p_n]$, k)
   Array $L[L_0, ..., L_n]$ //Create two array: L and MP
   Array $MaxProfit[mp_0, ..., mp_n]$
   L[0]=0 //Initialize L[0]
   for i=1 to n, i++ do

```
            if(m[i]-k<m[1])       //If there is no last possible restaurant before i
                L[i]=0
            else       //There is a last possible restaurant
                L[i]=L[i-1]+1
                while m[L[i]]≤m[i]-k    //Find the last possible restaurant before i
                    L[i]=L[i]+1
                L[i]=L[i]-1    //We over adding 1 to L[i] so we recover it
        MaxProfit[0]=0    //Initialize MaxProfit[0]
        for i=1 to n, i++ do
            //Calculate the maximum profit at index i by comparing two cases: opening the i restaurant or not opening
            MaxProfit[i] = max(MaxProfit[i-1], MaxProfit[L[i]]+P_i)
        return MaxProfit[n]    //The maximum profit is the last element in MaxProfit array
```

(d) (1 point) Analyze the running time (give a Θ-bound AND justification).

Solution: Firstly, we have two for loops, each loop takes n time, so it's 2n.
Secondly, in the first for loop, we have a while loop. However, this while loop is only executed at a constant time (at most K times, K is a constant). For example, one of the worst case is like: M[1,2,3,4,8], k=4. So, the L will be [0,0,0,0,0,4]. In this case, the while loop is executed 4 times when i=5. Thus, the while loop takes constant time. The overall running time is $\theta(n)$.

2. (9 points) Consider the following problem:
Input: A list of n positive integers $a_1, a_2, \ldots, a_n$; a positive integer t.
Question: Does some subset of the a_i's add up to t? (You can use each a_i at most once.)

You will use the following subproblem definition and recurrence to implement a solution to this problem.

Our subproblems will be of the form $S(i, x)$, which is $True$ if some subset of the first i elements of this list add to x. Given a list of i elements and a target x, we consider the last element i. If the first $i-1$ elements can add to $x - a_i$, then we could add element a_i and reach our target with the first i elements. This is also true if the first $i-1$ elements can add to x directly without including a_i. If *either* of these were true, we could sum to x using a subset of the first i elements. Thus: $S(i, x) = S(i-1, x-a_i) \vee S(i-1, x)$.

You will implement this solution by completing the following tasks and adding code to the file `DynamicSum.java`. Note that your implementation should have the best running time possible using the given subproblem and recurrence!

- Determine what the base case(s) should be. Hint: you will need a `True` and a `False` base-case.
- Implement the solution using *iteration* by adding code to the `isSum` method.

- Implement the solution using *recursion and memoization* by adding code to the `isSumMem` method.
- Write a method that returns the actual subset by adding code to the `getSubset` method. Note that this method must call one of the methods you wrote for the solution to part (b) or (c) to "fill the table". This method must not change the overall running time of the algorithm given above (i.e., it shouldn't take longer in terms of O notation than the time to "fill the table").
- As a comment at the top of each of the three methods (`isSum`, `isSumMem`, `getSubset`), analyze the running time of your algorithm in terms of n and t (use O notation and briefly explain your answer).

3. (5 points) A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Note that unlike a subarray, a subsequence does not have to be contiguous. For example, T, T, T is a subsequence of the sequence above. However, order does matter so A, A, A, A, G, G, G is not a subsequence while G, G, A, A, A, A, G is. More formally, to get a subsequence you can delete some or no elements from the original sequence but you can not change the order of the elements.

Use **dynamic programming** to devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the *length* of the longest palindromic subsequence. Its running time should be $O(n^2)$. You will implement your solution by adding code to the `getLengthLongestPalin` method within the `PalindromicSequence.java` file.

Note that if your code does not implement a dynamic programming solution (there must be a table!) with the required running time $O(n^2)$ then you will receive no credit regardless of whether the test cases execute correctly.

(EXTRA CREDIT) Write a method that returns the actual longest palindromic subsequence by adding code to the `getLongestPalin` method. Note that this method must call the `getLengthLongestPalin` method you wrote (or a similar private method) to "fill the table." This method must not change the overall running time of the algorithm (i.e. it's running time must be $O(n^2)$).

4. **(PRACTICE)** A certain wood-processing plant has a machine that can cut a piece of wood into two pieces at any position. This operation incurs a cost proportional to the length of wood being cut, n, regardless of the location of the cut. Suppose, now, that you want to break the wood into many pieces; you're given the locations of all the cuts you need to make in some array M. The order in which the breaks are made can affect the total cost. For example, if you want to cut a 20-foot piece at positions 3 and 10, then making the first cut at position 3 incurs a total cost of 20+17 = 37, while doing position 10 first has a better cost of 20+10=30.

Give a dynamic programming algorithm that, given the locations of m cuts in a piece of wood of length n, finds the minimum cost of breaking it into $m + 1$ pieces.

Additional Programming Instructions:

Note that your code will be automatically run on a standard set of test cases. In order to ensure that you do not lose points, follow the instructions below.

- Your code must compile without any errors using the version of Java on the lab computers. If your code does not compile you will not receive any points for the assignment.
- Do not modify any of the methods signatures (i.e. name, return type or input type). Note that you are always welcome (and encouraged) to add additional methods but these will not be run directly by the test code.
- You are not allowed to use packages (e.g. no statement `package ...` at the top of your file).
- No extra folders or files in your submission. Zip up only the files you need to submit not the folder they are in.
- Your solution should not print anything unless explicitly instructed to.

Grading Criteria:

Your work will be graded on both correctness **and clarity**. Write complete and precise answers and show all of your work. Your pseudo-code and proofs should follow the guidelines posted on Canvas and discussed in class.

Evaluation Rubric for Problem 1:

Component	Requirement for Full Credit
Subproblem Definition (1pt)	The definition is clear. It includes full sentences and should be easily understood by any student in the class.
Recurrence (2pts)	The recurrence is clear, concise and correct. Make sure to include at least one base case! It is described in full sentences and should be easily understood by any student in the class. It includes a justification for WHY the recurrence is correct.
Pseudo-code (1pt)	The pseudo-code is clearly written following the guidelines discussed in class.
Running Time Analysis (1pt)	The solution gives both a bound and an explanation. Both are clearly written and correct (for the algorithm given!).

Evaluation Rubric for Problem 2:

Component	Description
Style & Documentation (1 pt)	I'll be watching for style issues as well as correct output. See the syllabus for some general style guidelines (e.g. your code should be well-documented).
Base Case (.5 pt)	It should be clear from the comments in your code what the base case(s) are for the above subproblem definition and recurrence.
Big-O Analysis (1.5 pts)	Each method should include an analysis. Make sure to justify your analysis as well as give a bound (i.e., explain where your bound comes from and why it is correct). Each method should have the best running time possible using the given subproblem and recurrence.
Correct Output on Test Cases* (6 pts)	Each of your three methods will be run on a standard set of test cases. Make sure to test your code thoroughly!

*Note that if your code does not implement the given dynamic programming solution with the appropriate technique (iteration or memoization) you will receive no credit regardless of whether the test cases execute correctly.

Evaluation Rubric for Problem 3:

Component	Requirement for Full Credit
Style & Documentation (1 pt)	I'll be watching for style issues as well as correct output. See the syllabus for some general style guidelines (e.g. your code should be well-documented).
Correct Output on Test Cases* (4 pts)	Your `getLengthLongestPalin` method will be run on a standard set of test cases. Make sure to test your code thoroughly!
EXTRA CREDIT (1pt)	Your `getLongestPalin` method will be run on a standard set of test cases. Make sure to test your code thoroughly! You will receive NO credit if your code does use the table filled by the `getLengthLongestPalin` method (or a similar private method) or if it changes the running time to not be $O(n^2)$.

*Note that if your code does not implement a dynamic programming solution with the required running time $O(n^2)$ then you will receive no credit regardless of whether the test cases execute correctly.