Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 4 (100 pts) **Summer 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solution:

- The solutions **should be typed**, we cannot accept hand-written solutions. Here's a short intro to **Latex**.
- In this homework we denote the asymptomatic *Big-O* notation by $\mathcal{O}$ and *Small-O* notation is represented as o.
- We recommend using online Latex editor **Overleaf**. Download the **.tex** file from Canvas and upload it on overleaf to edit.
- You should submit your work through **Gradescope** only.
- If you don't have an account on it, sign up for one using your CU email. You should have gotten an email to sign up. If your name based CU email doesn't work, try the identikey@colorado.edu version.
- Gradescope will only accept **.pdf** files (except for code files that should be submitted separately on Canvas if a problem set has them) and **try to fit your work in the box provided**.
- You cannot submit a pdf which has less pages than what we provided you as Gradescope won't allow it.

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 4 (100 pts) **Summer 2020, CU-Boulder**

Piazza threads for hints and further discussion

Piazza Threads
Question 1a
Question 1b
Question 1c
Question 1d
Question 1e
Question 2
Question 3

Recommended reading:

Dynamic Programming: Chapter 15 complete

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 4 (100 pts) **Summer 2020, CU-Boulder**

1. (65 pts) The sequence L_n of Lucas numbers is defined by the recurrence relation

$$L_n = L_{n-1} + L_{n-2} \tag{1}$$

with seed values $L_0 = 2$ and $L_1 = 1$.

(a) (14 pts) Consider the recursive top-down implementation of the recurrence (1) for calculating the n-th Lucas number L_n.

i. (8 pts) Write down an algorithm for the recursive top-down implementation in pseudocode.

Answer:

```
int lucasNum(int n){
//given #1
if(n == 0){
   return 2;
}
//given #2
    if(n == 1){
        return 1;
    }
    //recursively call function
    else{
        //use sequence of Lucas numbers to create the recurrence
        return (lucasNum(n - 1) + lucasNum(n-2));
    }
}
```

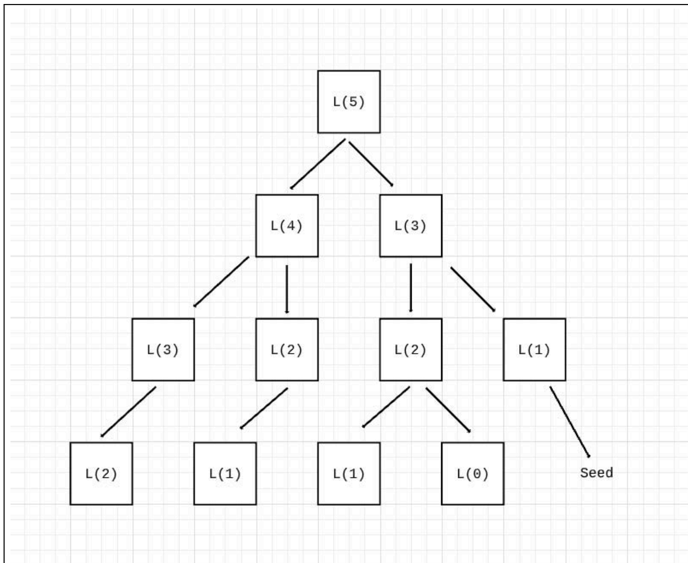
Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 4 (100 pts) **Summer 2020, CU-Boulder**

ii. (2 pts) Draw the tree of function calls to calculate L_5. You can call your function f in this diagram.



iii. (4 pts) Write down the recurrence relation along with the base case for the running time $T(n)$ of the algorithm.

$L_n = L_{n-1} + L_{n-2}$

The running time for L_n is $O(n)$ since the sequence is recursively called n times. Therefore, for a recurrence relation with the base cases of $L_0 = 2$ and $L_1 = 1$, the base case for the running time will be $O(1)$.

Answer: $T(n) = T(n-1) + T(n-2) + O(1)$

(b) (18 pts) Consider the dynamic programming approach "top-down implementation with memoization" that memoizes the intermediate Lucas numbers by storing them in an array $L[n]$.

i. (10 pts) Write down an algorithm for the top-down implementation with memoization in pseudocode.

Answer:

```
int lucasNum(int n){
    //declare array L
    //set all values to 0 (saying they are untouched)
    int L[] = 0;
    //given #1
    if (n == 0){
        return 2;
    }
    //given #2
    if (n == 1){
        return 1;
    }

    //check to see whether the Lucas Number has
    //already been included in the array
    if (L[n-1] != 0){
        //set temp value to index in array
        temp1 = L[n-1];
    }
    //if the value has not been included
    else {
        //set temp value of lucasNum(n-1)
        temp1 = lucasNum(n-1);
        //set value in array to n-1
        L[n-1] = temp1;
    }

    //check for n-2
    if(L[n-2] != 0){
        temp2 = L[n-2];
    }
```

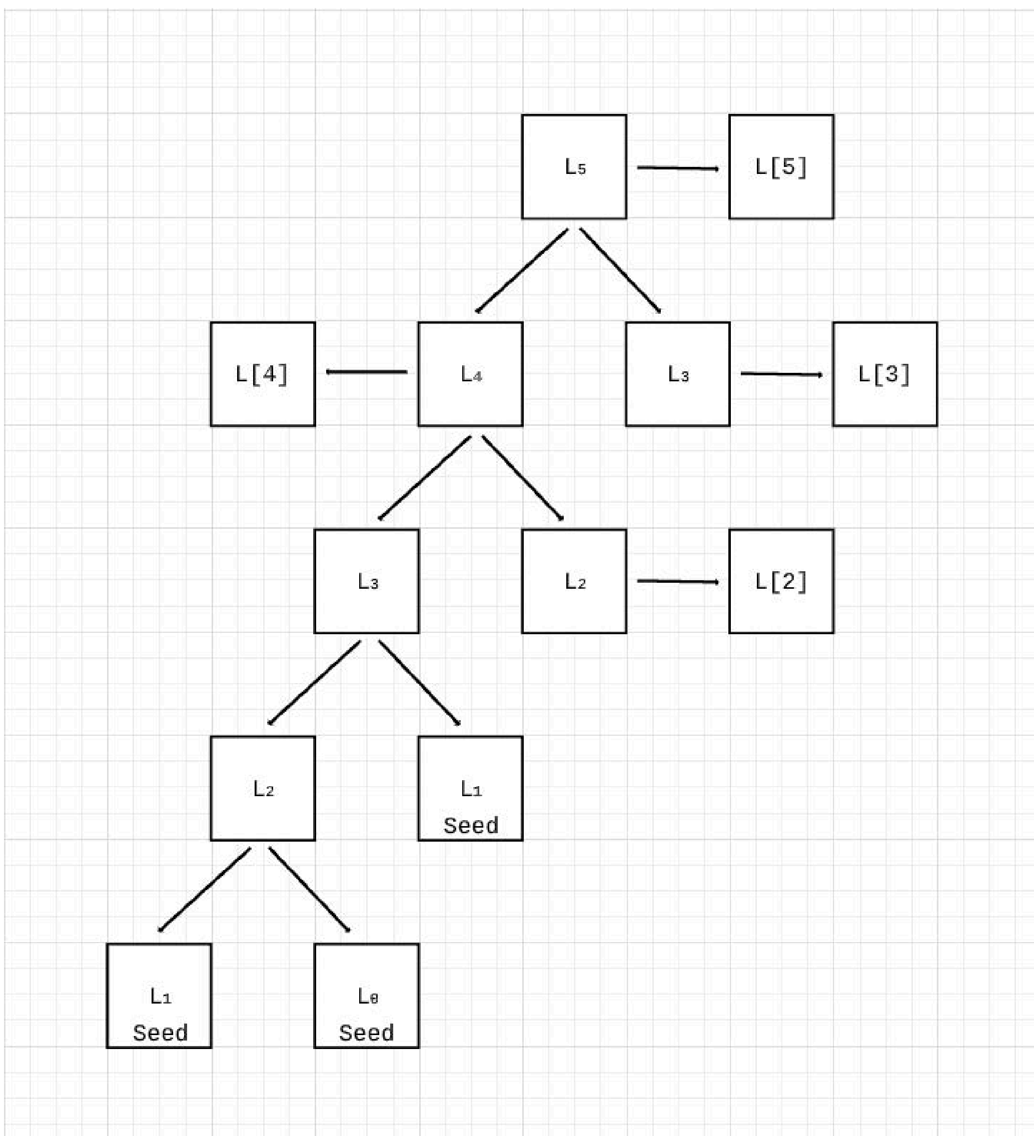
```
    else{
        temp2 = lucasNum(n-2);
        L[n-2] = temp2;
    }

    return temp1 + temp2;
}
```

ii. (2 pts) Draw the tree of function calls to calculate L_5. You can call your function f in this diagram.

If L[n] doesn't exist, the tree follow the recursion and finds the next $L_n = L_{n-1} + L_{n-2}$ to fill the array. It can then fill the tree with L[n].

Tree Shown Below



iii. (2 pts) In order to find the value of L_5, you would fill the array L in a certain order. Provide the order in which you will fill L showing the values.
Answer Shown Below

Call L_5 (splits into L_4 and L_3)

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
0	0	0	0	0	0

Call L_4 (splits into L_3 and L_2)

- When calling L_4, there is no change in the array. (Left branch from L_5)

Call L_3 (splits into L_2 and L_1)

- When calling L_3, there is no change in the array. (Left branch from L_4)

Call L_2 (splits into L_1 and L_0)

- When calling L_2, the seeds are changed. (Left branch from L_3)
- Changes L_0 and then L_1.

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	0	0	0	0

Call L_2 (L[2])

- When calling L_2, the array changes L[2].

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	0	0	0

Call L_3 (L[3])

- When calling L_3, the array changes L[3].

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	4	0	0

Call L_4 (L[4])

- When calling L_4, the array changes L[4].

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	4	7	0

Call L_5 (L[5])

- When calling L_5, the array changes L[5]. (Top of tree)

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	4	7	11

iv. (4 pts) Determine and justify briefly the asymptotic running time $T(n)$ of the algorithm.

Answer: T(n) = O(n)
The top down implementation with memoization has an asymptotic running time T(n) = O(n). In the algorithm, we search through the array. This data structure has a time complexity of $O(n)$ since it is called recursively n times.

(c) (16 pts) Consider the dynamic programming approach "iterative bottom-up implementation" that builds up directly to the final solution by filling the L array in order.

i. (10 pts) Write down an algorithm for the iterative bottom-up implementation in pseudocode.

Answer:

```
int lucasNum(int n){
    //declare array and set all values to 0
    int L[] = 0;
    //given #1
    L[0] = 2;
    //given #2
    L[1] = 1;

    //iterative loop to fill array
    for(int i = 2; i <= n; i++){
        //use sequence for Lucas numbers
        //use values from array to solve
        L[i] = L[n-1] + L[n-2];
    }
    //return value of L[n]
    return L[n];
}
```

ii. (2 pts) In order to find the value of L_5, you would fill the array L in a certain order using this approach. Provide the order in which you will fill L showing the values.
Answer Shown Below

Declare given values L[0] and L[1]

- L[0] = 2
- L[1] = 1

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	0	0	0	0

First run through loop (i = 2)

- L[2] = L[2 – 1] + L[2 – 2] = L[1] + L[0] = 3

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	0	0	0

Second run through loop (i = 3)

- L[3] = L[3 – 1] + L[3 – 2] = L[2] + L[1] = 4

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	4	0	0

Third run through loop (i = 4)

- L[4] = L [4 – 1] + L[4 – 2] = L[3] + L[2] = 7

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	4	7	0

Fourth run through loop (i = 5)

- L[5] = L[5 – 1] + L[5 – 2] = L[4] + L[3] = 11

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]
2	1	3	4	7	1

Returns L[5] = 11

iii. (4 pts) Determine and justify briefly the time and space usage of the algorithm.

Answer: T(n) = O(n)
The iterative bottom-up implementation has an asymptotic running time T(n) = O(n). The algorithm uses one for loop which executes n times to find L_n for each n. It then inserts the value into an array which has a time complexity of $O(n)$. This gives the algorithm a time complexity of $O(n)$.
The algorithm uses int n, int L[], and int i. The integers int n and int i use 4 bytes while int L[] uses 4n bytes. The space complexity is therefore 4 + 4 + 4n. The highest order of n in the equation is 'n' so the space complexity is $O(n)$.

(d) (7 pts) If you only want to calculate L_n, you can have an iterative bottom-up implementation with $\Theta(1)$ space usage. Write down an iterative algorithm with $\Theta(1)$ space usage in pseudocode for calculating L_n. There is no requirement for the runtime complexity of your algorithm. Justify your algorithm does have $\Theta(1)$ space usage.

Answer:

```
int lucasNum(int n){
    //declare given values/base cases
    int x = 2; //when i = 0
    int y = 1; //when i = 1
    int result = 0;
    //iterative loop to solve sequence for
    //increasing n
    for(int i = 2; i <= n; i++){
        result = x + y;
        x = y;
        y = result;
    }
    //return value of result
    return result;
}
```

This algorithm has a space complexity of $\Theta(1)$ because all variables are integers which use 4 bytes of space. Therefore, the space complexity is constant throughout all runs of the algorithm and every input.

(e) (10 pts) In a table, list each of the four algorithms(as part of (a), (b), (c), (d)) as rows and in separate columns, provide each algorithm's asymptotic time and space requirements. Briefly discuss how these different approaches compare, and where the improvements come from.

	(a)	(b)	(c)	(d)
time	$O(n)$	$O(n)$	$O(n)$	$O(n)$
space	$O(1)$	$O(n)$	$O(n)$	$O(1)$

For (a), the top-down implementation, the function lucasNum is called recursively n times, therefore yieleding a time complexity of $O(n)$. The space complexity for (a) is simply O(1) since all variables are integers and therefore using a constant 4 bytes of space each.
In (b), the top-down implementation with memoization, the function lucasNum is called recursively n times, like (a). This yields a time complexity of $O(n)$. The space complexity is $O(n)$ since the algorithm uses 1 array that uses 4n bytes of space each time. Therefore, the space complexity is O(n).
For the algorithm in (c), a bottom-up implementation, L[n] is found through a for loop. The for loop has a time complexity of $O(n)$ since n is called once for each run of the algorithm. Therefore, the algorithm as a whole has a time complexity of $O(n)$. Like (b), (c) uses 1 array that takes up 4n bytes of space and yields a space complexity of $O(n)$.
The bottom-up implementation in (d), was written to have a space complexity of $\Theta(1)$. This is done by only using integers instead of arrays like (c). The algorithm uses an iterative for loop to solve for L_n yielding a time complexity of $O(n)$ like the other 3 algorithms.
The use of arrays in (b) and (c) increase the space complexity through the use of arrays. (a) and (b) recursively call n while (c) and (d) use a for loop to use n to find L_n.

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 4 (100 pts) **Summer 2020, CU-Boulder**

2. (10 pts) Consider the following DP table for the Knapsack problem for the list $A = [(4,9),(1,6),(3,3),(5,12),(6,9)]$ of (weight, value) pairs.
The weight threshold $W = 10$.

- Fill in the values of the table.
- Draw the backward path consisting of backward edges and do not draw (or erase them) the edges that are not part of the optimal backward paths.

(a) (6 pts) Fill the table with the above requirements (You can also re-create this table in excel/sheet or on a piece of paper and add picture of the same).

Weight	Value	items_considered	0	1	2	3	4	5	6	7	8	9	10
-	-	no items											
4	9	A[0..0]											
1	6	A[0..1]											
3	3	A[0..2]											
5	12	A[0..3]											
6	9	A[0..4]											

Weight	Value	items_considered	0	1	2	3	4	5	6	7	8	9	10
-	-	no items	0	0	0	0	0	0	0	0	0	0	0
4	9	A[0..0]	0	0	0	0	9	9	9	9	9	9	9
1	6	A[0..1]	0	6	6	6	9	15	15	15	15	15	15
3	3	A[0..2]	0	6	6	6	9	15	15	15	18	18	18
5	12	A[0..3]	0	6	6	6	9	15	18	18	18	21	27
6	9	A[0..4]	0	6	6	9	15	18	18	18	18	21	27

d = diagonal

u = up

Weight	Value	items_considered	0	1	2	3	4	5	6	7	8	9	10
-	-	no items	0	0	0	0	0	0	0	0	0	0	0
4	9	A[0..0]				d							
1	6	A[0..1]				u	d						
3	3	A[0..2]						u					
5	12	A[0..3]						u					d
6	9	A[0..4]											u

Backtracking: Cell[row][] (row of no items is row 0)
Starting cell = [5][10] (last cell)

Current cell = [5][10] = 27
Above cell = [4][10] = 27
Previous cell = [5][9] = 21
Since the above cell has the same value, we can conclude that the fifth item (6,9) is not used in our optimal subset. We move up as a result.
Current cell = [4][10] = 27
Above cell = [3][10] = 18
Previous cell = [4][9] = 21
Since the values are different, the fourth item (5,12) is used in the subset.
The value of the fourth item is 12 so we subtract 12 from the current cell value (27 - 12) to get 15. We now search for the first 15 in our current row. This exists in [4][5].

Current cell = [4][5] = 15
Above cell = [3][5] = 15
Previous cell = 9
Since the above cell has the same value, we move up.

Current cell = [3][5] = 15
Above cell = [2][5] = 15
Previous cell = [3][4] = 9
Since the above cell has the same value, we can conclude that the third item (3,3) is not used in our optimal subset. We move up as a result.

Current cell = [2][5] = 15
Above cell = [1][5] = 9
Previous cell = [2][4] = 9
Since the values are different, the second item (1,6) is used in the subset.
The value of the second item is 6 so we subtract 6 from the current cell value (15 - 6) to get 9. We now search for the first 9 in our current row. This exits in [2][4].

Current cell = [2][4] = 9
Above cell = [1][4] = 9
Previous cell = [2][3] = 6

Since the above cell has the same value, we move up.

Current cell = [1][4] = 9
Above cell = [0][4] = 0
Previous cell = [1][3] = 0
Since both cells are 0, the first item (4, 9) is also included in our optimal subset. Therefore we subtract the value of the first item, 9, from our current cell value (9 - 9) to find 0. This completes the backtracking.

(b) (2 pts) Which cell has the optimal value and what is the optimal value for the given problem?

Answer: Cell[12][10] has the optimal value which is 27 (row item (5,12) and column 10). The maximum weight that can be carried with a weight threshold of 10 using the values in the list is 27.

Item 1: (1,6) $\rightarrow$ provides the largest value to weight ratio. By adding this item, the possible weight threshold is now reduced to 9 and the value is 6.
Item 2: (5,12) $\rightarrow$ provides the second largest value to weight ration. By adding this item, the possible weight threshold is now reduced to 4 and the value is increased to 18.
Item 3: (4, 9) $\rightarrow$ provides the same weight as the remaining weight threshold. By adding this item, the possible weight threshold is met and the value is increased to 27.
Therefore, the optimal value for the given problem is 27 which lies in cell[12][10].

(c) (2 pts) List out the optimal subset and provide it's weight and value.

Answer: Optimal Subset = $\{(4,9),(1,6)(5,12)\}$ with a total weight of 10 and value fo 27.

Name:	Pourna Sengupta
ID:	109086577

3. (25 pts) Given an array of n size, the task is to find the longest subsequence such that the **absolute difference** between two adjacent values in the sequence is odd and less than or equal to 5. i.e the **absolute difference** between the adjacent elements is one of the values from the set $\{1, 3, 5\}$
For the definition of subsequence click here

Example 1:
Input: $\{10, 30, 5, 8, 27, 1, 4, 9, 14, 17\}$
output: 6
Explanation: Here the longest sequence satisfying the above condition will be $\{10, 5, 8, 9, 14, 17\}$ having a size of 6

Example 2:
Input: $\{10, 30, 6, 9, 27, 22, 20, 19\}$
output: 4
Explanation: There are several sequences of length 4 one such sequence is $\{30, 27, 22, 19\}$ having a size of 4

(a) (5 pts) State the base case and recursive relation that can be used to solve the above problem using dynamic programming.

(b) (10 pts) Write down well commented pseudo-code or paste real code to solve the above problem.
Answer:

```
int longestSeq(int S[]){
    \\declare array
    \\set all values of array equal to 0
    int seq[] = 1;

    //compare values in S[]
    for(int i = 1; i < S.length(); i++){
        for(int j = 0; j < i; j++){
            //compare element i of array to all previous elements
            if((S[i] == S[j] + 1) || (S[i] == S[j] - 1)){
                //compare seq[i] to seq[j] + 1 to find max value
                if(seq[i] < seq[j] + 1){
                    //if seq[j] + 1 is greater
                    //update seq[i]
                    seq[i] = seq[j] + 1;
                }
            }
        }
    }
    for(int k = 0; k < S.length(); k++){
        //declare counter for size of array
        int count = 1;
        //compare count to max value in seq
        if(count < seq[k]){
            //the max value is the
            //largest possible
            //array size
            count = seq[k];
        }
    }
    return count;
}
```

(c) (5 pts) Discuss the space and runtime complexity of the code, providing necessary justification.

Answer: Time complexity: $O(n^2)$ and Space complexity: $O(n)$
The algorithm runs through two for loops. Therefore to complete each loop of i, it must complete all loops of j. Therefore, the time complexity is $O(n^2)$. The algorithm uses int S[], int seq[], int i, int j, int k, and int count. The integers i, j, k, and count all use 4 bytes of space. The arrays S[] and seq[] use 4n bytes of space. The total space needed for the algorithm is 2(4n) + 4(4) with the highest order of n being 'n'. Therefore, the space complexity is $O(n)$.

(d) (5 pts) Show how you can modify your pseudo-code or real code to return an optimal subsequence (if the problem has multiple optimal subsequences as part of it's solution, it is sufficient to return any one of those).

4. ***Extra Credit (5% of total homework grade)*** *For this extra credit question, please refer the leetcode link provided below or click here. Multiple solutions exist to this question ranging from brute force to the most optimal one. Points will be provided based on Time and Space Complexities relative to that of the most optimal solution.*

   Please provide your solution with proper comments which carries points as well.

   https://leetcode.com/problems/regular-expression-matching/

```
class Solution {
public:
    bool isMatch(string s, string p) {
        //declare and set variables for string lengths
        int a = s.length();
        int b = p.length();


        //if p is empty return empty s
        if(b == 0){
            return (a == 0);
        }

        //delcare new bool table for results
        bool matches[a + 1][b+1];
        for(int n = 0; n < a; n++){
            for(int m = 0; m < b; m++){
                matches[n][m] = false;
            }
        }

        //set matches[0][0] so that empty p and s can match
        matches[0][0] = true;

        //for loop where '*' is the only p that can match with
        //an empty s
        for(int i = 1; i <= b; i++){
            //check to see where p is '.'
            if(p.at(i-1) == '.'){
                //set all elements of p in matches to this value
```

```
                matches[0][i] = matches[0][i-1];
            }
        }

        //fill the table using bottom-up implementation
        for(int j = 1; j <= a; j++){
            for(int k = 1; k <= b; k++){
                //'*' either shows an empty sequence or matches
                //with element in input
                if(p.at(k-1) == '*'){
                    //set matches to the next character
                    matches[j][k] = matches[j][k-1] ||matches[j-1][k];
                }
                //now the characters match
                else if(s.at(j-1) = p.at(k-1)){
                    //set matches to previous character
                    matches[j][k] = matches[j-1][k-1];
                }
                else{ //characters do not match
                    matches[j][k] = false;
                }
            }
        }

        return matches[a][b];
    }
};
```