COT 4400 Analysis of Algorithms

Project 2: Dynamic Programming

Group 1

Donald Christensen
Pravin Henderson
Jialin Wang
Boyang Wu

1. How you can break down a problem instance of aligning a set of n teeth (a1, a2, . . ., an) with a set of m teeth (b1, b2, . . ., bm) into one or more smaller instances? Your answer should include how you calculate the minimum height for the original problem based on the solution(s) to the subproblem(s).

(a) The sub-problems are to add each pair from array1 and array2, and the largest sum of those pairs would the absolute maximum min-height that could be used to recurse through the teeth.

(b) Next, an intermediate minimum height would be calculated by adding the maximum of array1 and the minimum of array2, then doing the reverse by adding the minimum of array1 and the maximum of array2. Whichever sum is the greatest will be our absolute minimum min-height that is used to recurse through the teeth, and our final min height will never be lower than this number, and at most be this number

With each additional array shift that may be required to bring the teeth closer together, we have to recheck the absolute maximum value obtained from (a) to see if it decreased, until we either get to the absolute minimum obtained from (b), or if shifting the arrays will no longer decrease the min-height value.

2. What are the base cases of this recurrence?

The base case for the recurrence is the height for the first pair of the two sets of teeth. We initially set our min-height to be 0 at the start, and it will increase as we start adding array pairs and finding larger numbers.

Then, we set all values in the 2D matrix (used to compute the maximum value between array pairs) to a sentinel value of -1 since the sums between the pairs of array values should never be lower than 0.

Now when we go through the recursion, any value greater than our current min-height would be set to a value of -2, indicating that we do not need to go through that branch of the recurrence.

3. What data structure would you use to recognize repeated problems? You should describe both the abstract data structure, as well as its implementation.

We would use a 2D map (matrix/2D array) which is used to store the heights between the pairs of array values. Whenever we calculate a value greater than our current min-height, that number is set to -2 which means that we will never need to go any further through that recursion branch. From the dynamic programming portion of question 1 we see that we will always start at a larger min-height, and then try to go down to the lowest possible min-height value, so we will never need any values that end up being set to -2, and we can only change more values to -2 whenever our min-height gets updated.

4. Give pseudocode for a memoized dynamic programming algorithm to find the minimum height when aligning extensible teeth (a1, a2, . . ., an) and (b1, b2, . . ., bm).

```
Input vector data1:
Input vector data2:
vector<vector<data>> jaw:
Int maxData1:
Int maxData2:
Vector minSave;
getMinimumMaximum(vector MinSave){
      Priority_queue pq, pq2;
      while(int i != 2){
            If (data1.size < j)
                  pq.insert(data1.at(j));
                  J++;
            if(data2.size < k)
                  pq2.insert(data1.at(k);
                  K++;
      }
      maxData1 = pq.top();
      maxData2 = pq2.top();
      If maxData1 >= maxData2:
            Max = maxData1;
            Min = Max
            for(int i = 0; i < data2.size; i++){
                  If data2.at(i) < Min:
                  Min = data2.at(i);
            }
      If maxData1 < maxData2:
            Max = maxData2;
            for(int i = 0; i < data.size; i++){
                  If data.at(i) < Min:
                  Min = data.at(i);
            }

      Make pair Max and Miin = p1
      minSave.push_back(p1);
}
```

```
Input vector<int> A1;                                                    O(1)
Input vector<int> A2;                                                    O(1)
vector<vector<int> Mat;                                                  O(1)
Path = {};                                                               O(1)
Start = (0,0);                                                           O(1)
While Path does not contain [A1[n],A2[n]]:{                              O(m*n){
      GoRight = Max - Mat[1+x][y];                                       O(1)
      GoDown = Max - Mat[x][1+y];                                        O(1)
      GoCenter = Max - Mat[x+1][y+1];                                    O(1)
      Path.push(CompareMax(GoRight, GoDown, GoCenter, x, y));            O(1)
      if(Path.peakBottom() == (0,0)){                                    O(1)
            while(Path.length() > 0){                                    O(n)
                  Path.pop();                                            O(1)
            }
            Path.Push((0,0));                                            O(1)
      }
      Int x = path.x;                                                    O(1)
      Int y = path.y;                                                    O(1)
}                                                                                }
Return Path{};                                                           O(1)
End;                                                                     O(1)

CompareMax(GoRight, GoDown, GoCenter, x, y){                             O(1)
      If GoCenter == 0:                                                  O(1)
            Return (x+1,y+1);                                            O(1)
      If GoCenter > 0:                                                   O(1)
            If GoDown >= 0:                                              O(1)
                  If GoRight >= 0:                                       O(1)
                        if GoCenter <= (GoDown && GoRight);              O(1)
                              Return (x+1, y+1);                         O(1)

      If GoRight >= 0:                                                   O(1)
            If GoDown >= 0;                                              O(1)
                  If GoCenter >= 0;                                      O(1)
                        if GoRight <= GoDown && GoCenter);               O(1)
                              Return (x+1, y);                           O(1)
      If GoDown >= 0:                                                    O(1)
            If GoRight >= 0;                                             O(1)
                  If GoCenter >= 0;                                      O(1)
                        if GoDown <= GoRight && GoCenter);               O(1)
                              Return (x, y+1);                           O(1)
      if GoRight && GoDown && GoCenter < 0:                              O(1)
            Return(0,0);                                                 O(1)
```

5. What is the worst-case time complexity of your memoized algorithm?

Absolute worst-case time complexity will be n*m (size of array 1 and size of array 2) since it will mean we have to go through the entire n*m 2D matrix. This will not happen since we calculate the minimum height ahead of time, so will always choose values either directly in the diagonal or closest to the minimum height.

Here is a trace of how our algorithm finds the path in the 2D matrix. The blue cells on the side are the array values. The red cells are the cells that it goes through in the path. The yellow cells are the values that the algorithm sees but decides not to go to. The rest of the white cells are filled with -1 as a sentinel value. The rightmost column and bottommost row are used as empty spaces, so our algorithm does not have issues traversing the matrix.

array	index	0	1	2	3	4	5	6	7	8
index	element	6	4	1	3	5	4	7	2	2
0	2	8	6	3	-1	-1	-1	-1	-1	-1
1	7	13	11	8	9	-1	-1	-1	-1	-1
2	4	-1	-1	5	7	9	-1	-1	-1	-1
3	5	-1	-1	-1	8	10	-1	-1	-1	-1
4	3	-1	-1	-1	6	8	7	-1	-1	-1
5	1	-1	-1	-1	-1	6	5	8	3	-1
6	4	-1	-1	-1	-1	-1	8	11	6	-1
7	6	-1	-1	-1	-1	-1	-1	-1	8	-1
8	6	-1	-1	-1	-1	-1	-1	-1	-1	-1

6. Give pseudocode for an iterative algorithm to find the minimum height when aligning extensible teeth (a1, a2, . . ., an) and (b1, b2, . . ., bm). This algorithm does not need to have a reduced space complexity relative to the memoized solution.

```
Input A1[];
Input A2[];
Mat[A1][A2];
Path = {};
Start = (0,0);
While Path does not contain [A1[n],A2[n]]:{
      GoRight = Max - Mat[1+x][y];
      GoDown = Max - Mat[x][1+y];
```

```
      GoCenter = Max - Mat[x+1][y+1];
      Path.push(CompareMax(GoRight, GoDown, GoCenter, x, y));
      if(Path.peakBottom() == (0,0)){
            while(Path.length() > 0):
                  Path.pop();
            Path.Push((0,0));
      }
      Int x = path.x;
      Int y = path.y;
}
Return Path{};
End;

CompareMax(GoRight, GoDown, GoCenter, x, y){
      If GoCenter == 0:
            Return (x+1,y+1);
      If GoCenter > 0:
            If GoDown >= 0:
                  If GoRight >= 0:
                        if GoCenter <= (GoDown && GoRight);
                              Return (x+1, y+1);
      If GoRight >= 0:
            If GoDown >= 0;
                  If GoCenter >= 0;
                        if GoRight <= GoDown && GoCenter);
                              Return (x+1, y);
      If GoDown >= 0:
            If GoRight >= 0;
                  If GoCenter >= 0;
                        if GoDown <= GoRight && GoCenter);
                              Return (x, y+1);
      if GoRight && GoDown && GoCenter < 0:
            Return(0,0);
```

7. Can the space complexity of the iterative algorithm be improved relative to the memorized algorithm? Justify your answer.

Yes, since we only need to know the next diagonal, right, and down cells from our current cell, we only need to calculate and store those values when we get there, meaning that we do not need to store values for the entire 2D matrix as we did with the memoized algorithm. Since we only need the previous cell values, we don't need a full n*m sized matrix to store all the values. Large arrays, and thus large matrices for memoized/non-ideal iterative solutions would benefit the most from this.

8. Give pseudocode for an algorithm that identifies which teeth to extend in order to achieve the minimum height. Your algorithm may be iterative or recursive.

Our code goes through the path of the 2D matrix, and since it goes through the matrix and does not use an extend function, however if we were to create a teeth extension function it may look similar to the solution below. The function takes a maximum allowed gap and a minimum allowed gap then compares which teeth need to be extended to create an optimum gap.

Something that we noticed from the example in the project guidelines was that the extended teeth matching was not optimal, as there was a position where 1 and 4 match up, creating a 3 space gap in teeth, whereas if the 3 was shifted instead to match up in 3 and 4, it would only create a 1 space gap. We understood that the project did not necessarily care about minimizing the teeth gaps, and only the final minimum height for the whole array pairs, but it would make sense to reduce gaps in the final algorithm, which was what we decided to work towards.

```
findMinGap(maxGap, gap, bot jaw, top jaw){
      Check current position and compare to allowed gap:
      If current position > gap:
            Compare bottom jaw [position -1] and top Jaw for less than gap &&
< maxGap:
            Compare top jaw [position + 1] and bottom jaw for less than gap
&& < maxGap:
                  Extend bottom jaw[position -1];
                  Extend top jaw [position + 1];
                  Return True;
            Compare top jaw[ position -1][position] and bottom jaw for less
            than gap && < maxGap:
            Compare bottom jaw [position + 1] and top jaw for less than gap
            && < maxGap:
                  Extend top jaw[position -1];
                  Extend bottom jaw[position + 1];
                  Return true;

      Else
            Return false;

}

findMax(Int arr1, int arr2, &max){
      Int lowA1 = max;
      Int maxA1 = 0;
      Int lowA2 = max;
      Int maxA2 = 0;
```

```
	for(int i = 0; i < arr1.size; i++){

		If( arr1[i] > maxA1)
			maxA1 = arr1[i];
		if(arr2[i] > maxA2)
			maxA2 = arr2[i];
		if(arr1[i]  <  lowA1 && arr1[i] >= 0)
			lowA1 = arr1[i];
		if(arr2[i]  <  lowA2 && arr2[i] >= 0)
			lowA2 = arr2[i];
	}
	If( (maxA1 + lowA2)  < ( maxA2 + lowA1) ){
		Max = maxA1 + lowA2;
	}
	Else
		Max = maxA2 + lowA1;
}
```