b.

$$R(i,j,k) = \begin{cases} P(j,k) + C(k,i) & when\ i\ =\ m, base\ case \\ R(i+1,i,i)\ +\ R(i+1,m,m)\ +\ O(1) & when\ j\ =\ m\ =\ number\ of\ jobs \\ \begin{cases} when\ i\ not\ overlap\ k \begin{cases} R(i+1,m,m) + P(j,k) + C(k,i) & when\ adding\ i\ make\ shift > maxShiftLength \\ R(i+1,m,m) + P(j,k) + C(k,i) + R(i+1,j,i) + R(i+1,j,k) + O(1) & otherwise \end{cases} \\ when\ i\ overlaps\ k \begin{cases} R(i+1,m,m) + P(j,k) + C(k,i) & when\ adding\ i\ make\ shift > maxShiftLength \\ R(i+1,m,m) + P(j,k) + C(k,i) + R(i+1,j,k)\ +\ O(1) & otherwise \end{cases} \end{cases} \end{cases}$$

This is basically how my recursive algorithm works. R is the recursive function that takes 3 parameters i, j, k. P is the getProfit function, it calculates all the jobs' payment from j to k (both inclusive) and minus the cost from start date of j to end day of k. C is getCompensate function, it calculates jobs' payment from k to i(both exclusive).

Worst case: I want to go with the one that brings the max recursive calls as much as possible, the worst computation happens when i doesn't overlap with k and never the range between start of j to end of i exceed maxShiftLength. So my worst case would be

1. I have m jobs and for any 2 of them, they do not overlap.
2. The maxShiftLenght happens to be the range from the start of the first job and end of the last job.

$$R(0,m,m) \begin{cases} R(1,0,0) \begin{cases} R(2,0,1) \begin{cases} R(3,0,2) \\ R(3,0,1) \\ R(3,m,m) + P(0,1) + C(1,2) \end{cases} \\ R(2,0,0) \begin{cases} R(3,0,2) \\ R(3,0,0) \\ R(3,m,m) + P(0,0) + C(0,2) \end{cases} \\ R(2,m,m) + P(0,0) + C(0,1) \begin{cases} R(3,2,2) \\ R(3,m,m) \end{cases} \end{cases} \\ R(1,m,m) \begin{cases} R(2,1,1) \begin{cases} R(3,1,2) \\ R(3,1,1) \\ R(3,m,m)\ +\ P(1,1) + C(1,2) \end{cases} \\ R(2,m,m) \begin{cases} R(3,2,2) \\ R(3,m,m) \end{cases} \end{cases} \end{cases}$$

Above is what my recursive cases would look like and it goes on and on, I omitted the constant time picking the max for each case. If we look at the pattern on each level for the number of recursive cases, we see 1, 2, 5, 13, 34, …. The sum sequence S(n) is 1, 3, 8, 21, 55, …. And we know that the sum sequence is a sub set of the actual Fibonacci sequence S(n) = F(2n). In each case we do Ω(1) time. And F(4n)/F(2n) ≡ phi^2 ≡ 1.618^2 ≡ 2.618 for all n > 0, so its approximate growth is 2.618^{n}. The time complexity would be ***Ω(2.618^{m}).***

d.

The worst case is the same to both recursive and dynamic approach. The getStartJob() always takes constant time and return the first jobs. In the lowest level(base case), the outer 2 loop are trying to permutate the possible ways getProfit being called. The possible ways are

00	11	22	…	m-1 m-1
01	12	…	m-2 m-1	
02	…	2 m-1		
…	1 m-1			
0 m-1				

The time complexity of getProfit would be (m + 1) * m / 2 of job payment counted and at most ((m + 1) * m / 4) * n days counted, if you look at the triangle with different colors of layers, you'll notice the outer layer takes m days, (0 m-1) is a day and (00) + (1 m-1) is another day. The inner layer will have at most m-2 days since it's not a complete day, the start and end are missing. It's a number sequence up to m, if m is odd then it's 1, 3, 5, …, m, if m is even then it's 2, 4, 6, …,m. Either way it sums to ((m + 1) * m / 4). The getProfit in the lowest level would be O((m + 1) * m / 2 + ((m + 1) * m / 4)n) = O(m^2n). getCompensate is a smaller triangle so ignore that.

After having the base case, we do the same thing from m-1 layer all the way up to the first layer. While in this stage, it's just some constant time comparison and constant time fetching values from the previous layer but the complexity decreases since you ignore the jobs after the job you are currently processing. To sum up the time complexity, $1^2n + 2^2n + \ldots + m^2n = (\mathrm{m}(\mathrm{m}+1)(2\mathrm{m}+1)/6) * \mathrm{n} =$ $\boldsymbol{O(m^3n)}$**.**