Q1 b)
The best case for my recursive algorithm is if each job is skipped, which is only the case if all jobs are greater than maxShiftLength. This time complexity would be Ω (m).

The worst case upper bound recurrence for my recurrence algorithm would be if the recursive call was made every time. Resulting in T(m) = 2T(m-1) + O(n), m > 0. Where at each recursive call the worst case is if job[i].length == n causing the cost calculation to be O(n).

Using the iterative method

k = 1
T(m) = 2T(m-1) + n
k = 2
T(m) = 2(2T(m-1-1) + n) + n
T(m) = 4T(m-2) + 3O(n)
k = 3
T(m) = 2(4T(m-2-1) + 3n) + n
T(m) = 8T(m-3) + 7n
k = 4
T(m) = 2(8T(m-3-1) + 7n) + n
T(m) = 16T(m-4) + 15n
k = k
$T(m) = 2^kT(m-k) + (2k-1) * n$

stops when m – k = 1. Therefore when m - 1 = k.
For T(m) in terms of m. ie. sub in k
$T(m) = 2^{m-1}T(m-(m-1)) + (2(m-1)-1) * n$
$T(m) = 2^{m-1}T(1) + (2m-2-1) * n$
$T(m) = 2^{m-1}T(1) + (2m-3) * n$
$T(m) = 2^{m-1}T(1) + 2mn - 3n$
Which simplifies to
$T(m) = 2^{m-1} + mn – n$
$T(m) = 2^{m-1} + n (m-1)$
And results in a total lower bound for worst case to be $\Omega(2^{m-1})$

Q1 d)

maximumProfitDynamic

Runs with 2 for loops. The first of which runs from 0 to m, and the second (nested inside the first) runs from i to m. Within these for loops it is required to calculate the profits of the jobs, which includes the costs and losses of hiring the worker. The worst case for each of the cost function is if each job is of length n (all days). This is because the costs function would loop through the entire costs list of length n resulting in $O(m/2 * n)$. The worst case for losses is if the gap inbetween each job is n which is only possible for m = 2 and not possible if the cost function is at it worst and can be ignored since $O(m) < O(m/2 * n)$. Thus, the upper bound time complexity on this function would be $O(m*(m/2)*n)$. This simplifies to $O((m^2/2)* n)$ and finally $O(m^2 * n)$.

getMaxSubProblems

Runs with 2 for loops. The first of which runs from 0 to m, and the second (nested inside the first) runs from i to m. Thus, the upper bound time complexity on this function would be $O(m*(m/2))$. This simplifies to $O(m^2/2)$ and finally $O(m^2)$. This function also calls arrays.sort which is a comparison-based sort and is known to have a worst case complexity of $O(m \log(m))$. So the total complexity is $O(m^2 + m \log(m))$ which simplifies to $O(m^2)$.

maximumSubProblems

Runs with 2 for loops. The first of which runs from 0 to shifts.length which will be m in all cases. The second runs nested inside the first also running from i to m. This builds an array of worst case length m (take all shifts), which will eventually be looped through to find the max. This results in a total time complexity of $O(m * (m/2) + m)$ which simplifies to $O(m^2)$.

Overall Time Complexity

This results in an overall time complexity of the summation of each function. Therefore, $O(m^2 * n) + O(m^2) + O(m^2)$ which simplifies to $O(m^2 * n)$ resulting in a total time complexity of $O(m^2 * n)$.