2a.

```
P[i,x]=
    If(i=1){
        If(x=u){
            return U[1]
            }
        if(x=s){
            return S[1]
        }
    }
    else if(x=u){
        return max{ P[i-1,s]-m+U[i], P[i-1,u]+U[i] }
    }
    else if (x=s){
        return max{ P[i-1,u]-m+S[i], P[i-1,s]+S[i] }
    }
```

2b.

The maximum profit you can make in the first ‘i’ days, starting day ‘i’ with ‘u’ is equal to the highest of 2 scenarios:

1. The maximum profit in the first ‘i-1’ days, starting day ‘i-1’ with ‘u’ plus the profit of selling ‘u’ on day ‘i’.
2. The maximum profit in the first ‘i-1’ days, starting day ‘i-1’ with ‘s’ plus the profit of selling ‘s’ on day ‘i’ minus the cost of switch – ‘m’.

The same logic can applied to find the maximum profit in the first ‘i’ days, when starting day ‘i’ with ‘s’, simply by swapping the ‘u’s and ‘s’s.
When ‘i’ is 1, the max profit is either U[1] or S[1] respectively.

2f.

Lines: 42-44 of DSAP2.java

```
for(int i=0; i<n; i++){
    op_1 = iterative(i,'u');
    op_2 = iterative(i,'s');

}
```

From this we can see that the function ‘iterative’ is run ‘2n’, times. Each call of the function, plus the function runtime essentially takes constant time. Therefore the time complexity of the iterative algorithm is O(n).

3a.

```
Highest_score_team(v){
    if(number_of_apprentices= 0){
        return magical_ability[v]
    }
    else{
        return max{
            magical_ability[v] +
            SUM(Highest_score_team(each_grand_apprentices)),
        SUM(Highest_score_team(each_apprentice))}
    }
}
```

Where ‘v’ is the root node of the tree.

3b.

The highest scoring team of a given subtree with a root at ‘v’ is the max of:

1. The sum of highest scoring teams of all the subtrees that have roots that are children of ‘v’.
2. The sum of the highest scoring teams of all the subtrees that have roots that are grandchildren of ‘v’, plus the magical ability of node ‘v’ itself.

When a subtree has no children, then the highest scoring team is just the magical ability of the node itself.

3d.

Memoization would be useful for this problem, as the highest scoring team, is calculated for almost every root node at least twice, once when the node is being calculated as a child, and once when it is being calculated as a grandchild.

4a.

```
Day_off(i,j){
        if(i=n){
                return min{ b[i], m[j]}
        }
        else return max{ min{b[i],m[j]}+Day_off(i+1,j+1), Day_off(i+1,0) }
}
```

4b.

In order to work out whether you should take the day off, you need to find out whether you ought to deliver sandwiches today and tomorrow, or just tomorrow and take today off, in order to maximise the number of delivered sandwiches. This can be repeated for every day until i=n, at which point you will never take day 'n' off as there is nothing to be gained.

4c.

You will run the Day_off() fomula with 'i' and 'j' initially both set to 0. i.e. Day_off(0,0).

4h.

Lines: 25-30 of DSAP4.java

```
for(int i = n-1; i>=0; i--){
        for(int j = i; j>=0; j--){
                iterative(i,j);
        }
}
```

From this we can see that the function 'iterative'-which takes constant time- is called n+(n-1)+(n-2)+(n-3)……1 times.
Which is the same as the sum from 1 to n, which equals $\frac{n(n+1)}{2}$. Therefore the time complexity of the algorithm is O($\frac{n(n+1)}{2}$).