Part 1:

(a)

x = number of activities

y = total risk budget + 1

Initialize a 2d array called memo with x rows and y columns. Initialize first column to all 0's.

```
rec (activities, memo, cur_activity, rem_budget):
        //if at boundary
        if cur_activity_num == null || rem_budget == 0:
                return 0
        //if in memo
        else if memo[cur_activity][rem_budget] != null:
                return memo[cur_activity][rem_budget]
        //if remaining budget is not enough for current activity
        else if rem_budget < cur_activity.risk:
                return rec(activities, memo, cur_activity – 1, rem_budget)
        //if current budget is enough, then we can 1. Take 2. Not take the current activity
        else:
                memo[cur_activity][rem_budget] =
                        max(rec(activities, memo, cur_activity – 1, rem_budget),
                           rec(activities, memo, cur_activity – 1, rem_budget -
                        cur_activity.risk) + cur_activity.fun_level)
                return memo[cur_activity][rem_budget]
```

Time complexity:

Let there be R rows and C columns. Then the time complexity will be O(R*C), because, by memoization, we will visit each cell at most once, and there are R*C cells. It's pseudo polynomial because R, representing different levels of risk, is a "magnitude."

(b)

I will add a trace method to find all the selected activities.

```
trace (memo, row, col)
    if row == 0:
        if memo[row][col] != 0:  //meaning the first item was selected
            add the activity of the current row to the result
    else if memo[row][col] == memo[row-1][col]:  //meaning current activity is not selected
        trace (memo, row-1, col)
    else: //current activity is selected
        add the activity of the current row to the result
        trace (memo, row-1, col – current_activity.risk)
```

runtime is not changed. This method will be called R times, one time each row, so this is O(R). And clearly O(R) + O(R*C) is still O(R*C).

Part 2:

Initialize a 2d array, with size n x 2, where n is the number of days. Column 0 of a row will be the accumulative minimum cost if we choose to stay at Maui on that day, and Column 1 will be the accumulative minimum cost if we choose to stay at Oahu on that day. For example, array[4][0] will be the minimum accumulative cost if we choose to stay at Maui on day 4.

```
//fill up the 2d array

For i = n to 0:

    array[i][0] = min(array[i-1][0]+mauiCost[i], array[i-1][1]+mauiCost[i]+transferCost)

    array[i][1] = min(array[i-1][0]+oahuCost[i]+transferCost, array[i-1][1]+oahuCost[i])

//trace back solution

trace(array, n-1, last_location)


trace(array, idx, current_location):

    result[idx] = current_location   //record current location

    if idx == 0: //if we have reached the end of array

        return

    else: //have not yet reached the end of array

        if current_location == “Maui”: //if we are currently at Maui

            if array[idx][0] == array[idx-1][0]+mauiCost[idx]: //previous day at Maui

                trace(array, idx-1, “Maui”)

            else: //previous day at Oahu

                trace(array, idx-1, “Oahu”)

        else: //if we are currently at Oahu

            if array[idx][1] == array[idx-1][0]+oahuCost[idx]+ transferCost: //previous day
                                                                                  at Maui

                trace(array, idx-1, “Maui”)

            else: //previous day at Oahu

                trace(array, idx-1, “Oahu”)
```

The pseudocode might look a little complicated, but the tracing technique is very similar to part 1. We simply use addition to see which one of the two previous locations can add up to exactly the current location's minimum cost.

Constructing the 2d array requires O(n), because we do a constant amount of work on each row (i.e. using the previous row's info to fill out current row's columns).

Tracing is also O(n), because we recursively call the trace method n times (n row and each row gets examined once, so n times in total)

So total runtime is O(n)