The time complexity analysis for number 1:

In order to fully understand the time complexity of this algorithm that me and my partner created, I have posted the entire code below and am going to continually refer to it in order to clearly show the time the code takes. Here is the code:

```
1) def matmult (A,x):
2)     n = len(x)
3)     b = np.arange(len(A))
4)     for i in range (0,len(A)):
5)         temp = 0
6)         for j in range (0,n):
7)             temp += A[i][j] * x[j]
8)         b[i] = temp
9)     return (b)
```

When looking at this code, one thing clearly shows. There lies a for loop within another for loop. The first for loop loops through the columns of the array being inputted while the second for loop loops through the rows. The number of rows is equal to the number of columns since A is an n x n matrix. This then implies that for each increment of the value of n by one for the rows, the columns loop through n amount of times. This further shows that the time complexity for both these for loops to be completed is n^2 because of what was just said in the previous sentence. Thus, for lines 4 to 8 to be completed, it will take n^2. Lines 2, 3, and 9 all take an amount of time that is much lower than n^2 so thus, the overall algorithm takes $O(n^2)$. In fact, it can also be argued that there will be a tighter bound. Since the input array A will always be an n x n array and the vector length is always n, both these for loops will always take n^2 amount of time no matter the input array or input vector as long as the conditions are met. The conditions are that the array A is of size n x n and the vector is of size n. Now, since we know that with these conditions, both these for loops will have to loop through all values of n and the number of rows equals the number of columns, this algorithm will always take an n^2 amount of time, no matter what. There is no best and worst case input for this algorithm since the best, worst, and average case are all the same. In all these cases, all the values of n have to be looped through. Thus, it can be seen that the original time complexity I have made of $O(n^2)$ can be rewritten with a tighter bound of $\theta(n^2)$.

The derivation of the algorithm, and its time complexity analysis for problem 3:

The derivation of the algorithm:

For problem 3 we have to use the divide and conquer technique to multiply an 2^k x 2^k Hardamard matrix t=with an arbitrary 2^k dimensional vector. By theory the length of the Hardamard matrix will never be less than 2 because the base case states that k > 0 implying that k = 1 is the lowest value of k that could be inputted. 2^1 = 2 meaning the minimum length of H is 2. The divide and conquer technique can solve this problem by splitting the arbitrary 2^k dimensional vector and the 2^k x 2^k Hardamard matrix have half. Then it multiplies the individual split Hardamard matrix and the split arbitrary vectors and combine them together to get the solution.

Following this theory we have to have a variable that will be equal to the length if the vector divided by 2. This variable will be the location for the split in the vectors. We would then have to copy the first half of the vector to an array called x1 and then copy the second half of the vector to another array called x2. We can do this using two for loops. We then check the base case if the length of the Hardamard matrix equal to 2. If the Hardamard matrix equals 2 we could multiply the Hardamard matrix with the two arrays we created for splitting the arbitrary vector. If the length of the Hardamard matrix is greater than 2 we would then recursively call the method 2 times. Each time we would take in the Hardamard and the vector x1 and x2. We then add one of the arrays which we obtained by the recursive methods to itself and the other array we add it to the inverse of itself. We then have to concatenate the 2 arrays we just obtained to get the solution.

Time complexity analysis:

This is how we deduced our algorithm to receive a T(n) equation:

```
1)  def hadmatmult(H,x):
2)      n = len(x)//2
3)      x1 = x[0:n]
4)      x2 = x[n:len(x)]
5)      if len(H) == 2:
6)          return np.array([(H[0,0]*x1[0] + H[0,1]*x2[0]),(H[1,0]*x1[0] + H[1,1]*x2[0])])
7)      else:
8)          temp = hadmatmult(H[0:len(x1),0:len(x1)],x1)
9)          temp2 = hadmatmult(H[0:len(x1),len(x1):len(x)],x2)
10)         temp3 = temp
```

11) temp4 =-1*temp2

12) created = temp + temp2

13) created2 = temp3 + temp4

14) final = np.concatenate((created,created2), axis = 1)

15) return final

We will now figure out how long each line of code takes in order to figure out our T(n) equation:

Line 1:	T(n)
Line 2:	1
Line 3:	n/2
Line 4:	n/2
Line 5:	1
Line 6:	1
Line 7:	1
Line 8:	T(n/2)
Line 9:	T(n/2)
Line 10:	1
Line 11:	1
Line 12:	1
Line 13:	1
Line 14:	1
Line 15:	1

Now after adding up all these parts together we get this:

We can assume that all the lines that take a constant time can be summed up to a constant c.

Thus,

$T(n) = n/2 + n/2 + T(n/2) + T(n/2) + c$

$= 2T(n/2) + n + c$

Since c is a constant time and n is growing faster than the constant time, it can be negligible and n + c = n. Thus our final T(n) equation looks like this:

$T(n) = 2T(n/2) + n$

Using the master theorem as shown below we were able to figure out the time complexity:

Let T(n) be defined by the non-negative integers by the recurrence:

T(n) = aT(n/b) + f(n) where a is greater or equal to 1, b is greater than 1, and f(n) is a function. From this equation and our T(n) equation that we have above, we can deduce that a = 2, b = 2,, and f(n) = n

The second asymptotic bound states that if f(n) = O($n^{log_b n}$), then T(n) = Θ($n^{log_b n}$log n).

If we were to substitute our values of a = 2, b= 2, and f(n) = n into f(n) = O($n^{log_b a}$), we will get:

n = O($n^{log_2 2}$)

n = O(n^1)

n = O(n)

Since, both sides are equal to each other, we can deduce that the time complexity it will take is:

T(n) = Θ($n^{log_b n}$log n)

= Θ($n^{log_2 2}$log n)

= Θ(n^1log n)

= Θ(n log n)

Thus, it can be deduced from the master theorem, that the time complexity analysis for this algorithm is Θ(n log n) since f(n) = O($n^{log_b a}$) for the constants is a = 2 which is ≥ 1, b = 2 which is >1,and the function f(n) = n.

Comments on the trends for problem 5:

The graphs that were created from our algorithms exemplify what we expected to happen. As proven by analyzing our algorithm our brute force method ended up being theta (n^2). Meanwhile, our time complexity algorithm for using recursion ended up being theta (n) as proved above. This can clearly be seen in our graph as our time complexity graph for brute force is going up at a rate of n^2 and the hadmatmult is going at the rate of n. As the value of k gets incremented, our matmult time complexity graph takes a longer time at a fast rate of n^2. Meanwhile, ask gets incremented by 1 for the hadmatmult time complexity graph, the time is increasing at a linear rate. The matmult time complexity graph is being incremented as a quadratic rate. These are the differences between these two graphs.