1) The while-true loop will run indefinitely since there is no condition to terminate out of the loop. Because this procedure does not halt, it is not an algorithm. Answer is b).

2) The while-true has a condition statement when i=10000000000, the loop will break out. i is incremented by one each time, so i will eventually reach 10000000000. Because the procedure will halt at some point, it is an algorithm. Answer is a).

3) The while-true loop in this example will also run indefinitely. There is an external_procedure call in the loop, but the loop does not do anything with its return value (if the procedure has one). Whatever code that's inside of the external_procedure has no affect on anything outside of the scope. Because this procedure does not halt, it is not an algorithm. Answer is b).

However, you could say the external_procedure could call the exit function that could terminate the entire program. In that case, the algorithm does stop. Or the external_procedure could do something else, but not call the exit function, in which case the algorithm does not halt. It may depend on the operating system. Based on this, we can say the answer is c).

4) The outer and inner loop both go from 1 to N. The other code inside the loop take constant time. So the computational complexity is $O(N^2)$. We can show our work with a double summation. We consider $v[i] = 0$ as 1 operation and $v[i] += A[i, j] * b[j]$ as 3 operations.

$$\sum_{i=1}^{i=N}(1 + \sum_{j=1}^{j=N} 3) = \sum_{i=1}^{i=N}(1 + 3N) = N(1 + 3N) = N + 3N^2 = O(N^2)$$

5) The computational complexity should also be $O(N^2)$ because of the nested for loops, even if the inner loop index start at j=i instead of j=1. We can show this by writing the summation.

$$\sum_{i=1}^{i=N}(1 + \sum_{j=i}^{j=N} 3) = \sum_{i=1}^{i=N}(1 + 3(N - i + 1)) = \sum_{i=1}^{i=N}(3N - 3i + 4) = 3N^2 - \frac{3N(N+1)}{2} + 4N$$

$$= \frac{(6N^2 - 3N^2 - 3N + 8N)}{2} = (\frac{3}{2}N^2 + \frac{5}{2}N) = O(N^2).$$

6) The computational complexity is $O(N)$. Since M is a symbolic constant, the inner loop is essentially another constant, so only the parameter N in the outer loop affects the complexity. We note that when N is large, the inner loop will not run when $i > M$. We can show the complexity is $O(N)$ by writing the summations and splitting it at M.

$$(\sum_{i=1}^{i=N-M+1} 1) + \sum_{i=1}^{i=M}(1 + \sum_{i=i}^{i=M} 3) = N - M + 1 + \sum_{i=i}^{i=M}(1 + 3(M - i + 1)) = N - M + 1 + \sum_{i=1}^{i=M}(3M - 3i + 4)$$

$$= N - M + 1 + 3M^2 - \frac{3M(M+1)}{2} + 4M = N + C_1 = O(N)$$

7)

```
def my_pow(x, m):
  if(m == 0):
    return 1
```

```
  if(m == 1):
    return x
  pow_half = my_pow(x*x, m//2)
  elif(m % 2 == 0):
    return pow_half
  else:
    return x * pow_half
```

1) (optional for HW 1)

2) The algorithm is $O(log_2(M))$. Each recursive call is M/2, until it reaches the function's base cases. We can see this from a few simple cases. my_pow(3, 16), the recursive function calls M = 8, 4, 2 1 and my_pow(39), M = 19, 9, 4, 2, 1, a log number of times. We can use an inductive argument for this. But more formally this algorithm is $T(M) = T(M/2) + O(1)$ and solving this recurrence we get $O(log_2(M))$.

8)

```
def find_start(li):
  lo = 0
  hi = len(li)-1
  mid = 0
  while(hi>=lo):
    mid = (hi+lo) //2
    if(li[mid] > li[hi]):
      lo = mid+1
    elif(li[mid] < li[hi]):
      hi = mid
    else:
      return mid
```

1) (optional for HW 1)

2) The algorithm is $O(log_2(N))$, where N is the length of the input list. The algorithm gets rid of half of the search space at each iteration by comparing middle index to the highest index. This is essentially same as binary search, which is $O(log_2(N))$.