Problem 1

a) Greedy Algo for coin change

```
def getMin(N):{
res = []
coins = [1, 5, 10, 25]
n = coins.length
for (i = n-1; i >= 0; i--) {
    while (N >= coins[i]) {
        N = N - coins[i]
        res.append(coins[i])
    }
}
return res;
}
```

Here in greedy algorithm, we try to get max possible coins of max. value. Here 25 is max value, then once N is less than 25 we go for 10 then 5 and then 1.

b) consider coin set [1, 5, 12] and for N = 15 we have to find min coins. Here by greedy approach we get. {12, 1, 1, 1} i.e. 4 coins.

But this is not optimal solution. Optimal solution is {5, 5, 5} i.e 3 coins

Problem 2

counterexample for greedy strategy.

i	1	2	3	4	5
p	5	20	33	36	40
p/i	5	10	11	9	8

for length 4 by greedy approach we get solution (3+1). since $i=3$ has largest density $\therefore$ value $= 33+5=38$

but optimal solution is to divide rod by length 2.

$\therefore$ 2 pieces of rod with length 2 will fetch value of (20+20) i.e. 40.

Problem 3.

To solve this problem we can sort the given array and then decide the number of intervals.

To decide intervals, choose 1st item from array and unit interval we have is $(x, x+1)$ where x is 1st item.

Look up in sorted array if next number fits in interval. If it does then go to next number, else we can start with new interval.

```
int getIntervals (float[] arr) {

    sort (arr);
    int count = 0;
    float interval = arr[0] + 1.
    for (int i = 1 ; i < arr.length ; i++) {
        if ( arr[i] < interval )
            continue ;
        else {
            count ++;
            interval = arr[i] + 1 ;
        }
    return count ;
}
```

In this algorithm, consider sort() function uses merge sort.
After sorting we perform linear check to count intervals. It will take O(n) time.
∴ Time complexity will be $O(n \log n) + O(n)$
i.e. $O(n \log n)$.

for given example, after sorting, we get

arr = [0.8, 1, 1.7, 2.3, 3.1, 3.6, 3.9, 4, 4.2, 4.7].

1st interval = (0.8, 1.8). which covers 0.8, 1, 1.7
2nd interval = (2.3, 3.3) which covers 2.3, 3.1.
3rd interval = (3.6, 4.6) which covers 3.6, 3.9, 4, 4.2
4th interval = (4.7, 5.7) which covers 4.7.

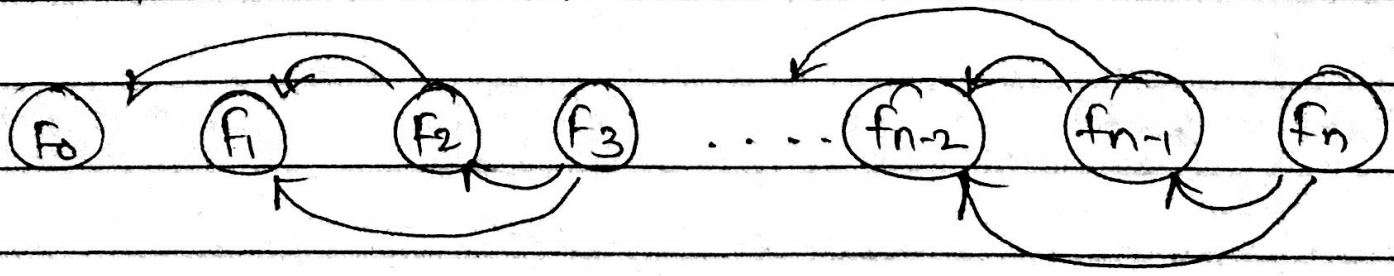
Hence count = 4 will be returned from given algorithm with time complexity of $O(n \log n)$.

Problem 4.

Consider following solution using bottom up approach.

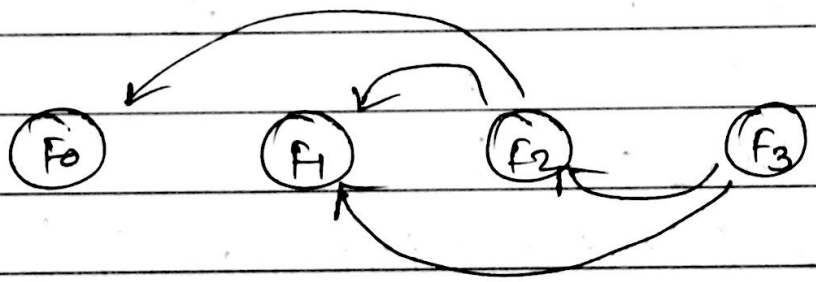
```
int getfib ( int n ) {
    int fib[] = new int [n+1];
    fib[0] = 0 ;
    fib[1] = 1 ;
    for ( int i = 2 ; i ≤ n ; i++) {
        fib[i] = fib[i-1] + fib[i-2] ;
    }
    return fib[n] ;
}
```

sub problem graph for n^{th} fibonacci num



considering the graph, to find n^{th} fibonacci number, we need n vertices and $(n-2) * 2$ edges.

e.g. for $n = 4$.



Here we have 4 vertices and 4 edges.

Problem 5

Given below is algorithm with complexity O(nW).

```
int knapsack (int W, int[] weights, int[] values) {
    int n = weights.length;
    int B[][] = new int[n+1][W+1];

    for (int i=0; i<n+1; i++)
        B[i][0] = 0
    for (int i=0; i<W+1; i++)
        B[0][i] = 0

    for (int i=0; i<=n; i++) {
        for (int w=0; w<=W.; w++) {

            if (weights[i-1] <= w)
                B[i][w] = max( B[i-1][w],
                    values[i-1] + B[i-1][w - weights[i-1]]

            else
                B[i][w] = B[i-1][w]
        }
    }
    return B[n][W];
}
```

v	w	i ↓ / W →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	1	0	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	3	2	0	0	3	4	4	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
5	4	3	0	0	3	4	5	7	8	9	9	12	12	12	12	12	12	12	12	12	12	12	12
8	5	4	0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	20	20	20	20	20
10	9	5	0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	21	22	23	25	26

Max value of items that can be kept in knapsack is 26 [2, 4, 5, 9].

Problem 6.

```
int getWays (int n) {
    int[] ways = new int[n+1];
    ways[0] = 0;
    ways[1] = 1;
    ways[2] = 2;
    ways[3] = 4;
    for (int i=4; i<=n; i++) {
     ways[i] = ways[i-1] + ways[i-2] + ways[i-3];
    }
    return ways[n];
}
```

Above algorithm computes value of ways to reach nth stair using bottom up approach of dynamic programming.

Here steps to reach 1st, 2nd and 3rd step are stored and they can be used to compute ways[n].