Practice problems: recursion, recursive function calls tree, memoization,

Some problems have points associated with the question. Please note that those serve as an example. The actual points allocated in your exam for a similar problem may be different.

If you have a hard time understanding recursion, see 'Training' under recursion from Visual Algo:
https://visualgo.net/training?diff=Medium&n=7&tl=0&module=recursion

P1. Write recursive implementations for:

- Find the minimum element in an array
- Add all the values in an array.
- Selection sort
- Insertion sort
- Binary search

P2. Even-at-even, odd-at-odd

a) (10 pts) Given an array, A, of strings and the number, N, of strings in the array, return one string that is the concatenation of all these strings. You can assume that the max length for the input strings is 20 and that at most 50 strings would be given as input. You can write helper functions.
You can write your own function to copy strings or use the C provided one. If you use the C library one, assume it copies a string character by character (not as a large block of memory).
b) What is the time complexity of your function? If the max length of a string and the max number of strings affect your time complexity, use L_max and N_max for them.

P3. Even-at-even, odd-at-odd

a) (10 pts) Write a **recursive** function that takes as argument an array and its length and returns:
   a. 1 if the elements at even positions are even and the elements at odd positions are odd. (The number 0 is considered even.)
   b. 0 – otherwise.

The function must use recursion. You can write auxiliary functions if needed.
Sample behavior:
- for [6, 23, 8, 7] it returns because at even positions (0 and 2) we have even numbers (6 and 8) and at odd positions (1 and 3) we have odd numbers (23 and 7).
- for [6, 23, 5, 7] it returns 0 (because at even index, 2, there is an odd number, 5).
- for an empty array it returns 1.

b) (2 points) Write the recursion formula for your function.

c) (2 points) What is the runtime of your function? (No justification needed.)

P4. Equal arrays

a) (12 pts) Write a **recursive** function `equals(int* A, int* B, int N)` that takes as argument two arrays and their length and returns:

a. 1 if the arrays are identical (have, position-wise, the same elements).
b. 0 otherwise.

The function must use **recursion** and should **NOT** have **ANY LOOP**. You can write auxiliary functions if needed.
You can assume that both arrays have the same length.
Sample behavior:
- equals([6, 23, 8, 7], [6, 23, 8, 7], 4) returns 1.
- equals([6, **23**, 8, 7], [6, **5** , 8, 7], 4) returns 0 (because A[1] ≠ B[1]).
- equals([ ], [ ], 0) returns 1.

b) (2 points) Write the recurrence formula for the **time complexity** of your function. Include the base case.

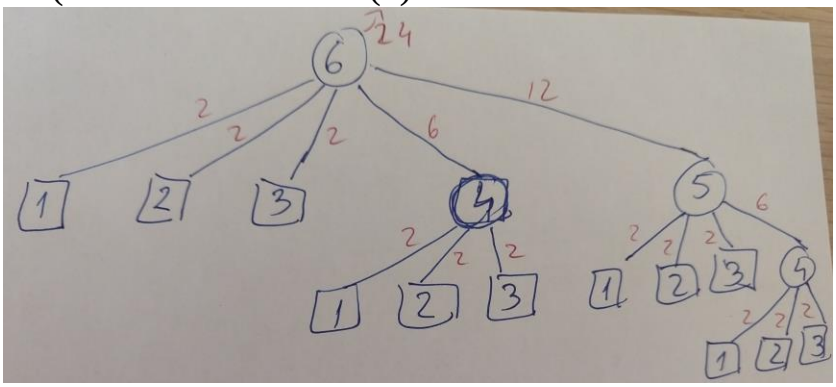
c) (2 points) Give the time complexity of your function as **Θ**. No justification needed.

P5. Consider this recursive function **foo(N):**

```
int foo1(int N){
    if (N <= 3) return 2;
    int total = 0;
     for(int r = 1; r < N; r++){
          total = total + foo1(r);
     }
     return total;
}
```

a) (6 points) Write the recurrence formula for this function (including the base cases) for N>=0. (You do NOT need to solve it.)
T(N) = c for all N≤3
T(N) = T(1)+T(2)+T(3)+…+T(N-1) + cN

b) (5 points) Draw the tree that shows the function calls performed in order to compute foo1(6) (the root will be foo1(6) and it will have a child for each recursive call.)



c) (3 points) Compute the result value returned by foo1(6). **foo1(6) = 24**

d) (10 points) Re-implement this function with memoization (i.e. use a solution array to look-up and store results of recursive calls).
   a. (7 points) Write the memoized recursive function.
   b. (3 points) In addition to showing the changes in the foo function, **also write the wrapper function** that calls the memorized **foo** function.

P6. Consider this recursive function **foo(N):**

```
int foo2(int N){
     if (N <= 3) return 2;
     int total = 0;
     for(int r = 1; r<=N; r++){
          total = total + r;
     }
     return total+foo2(N/2);
}
```

e) Write the recurrence formula for this function (including the base cases) for N>=0. (You do NOT need to solve it.)
   T(N) = c for all N≤3
   T(N) = T(N/2) + cN

f) Draw the tree that shows the function calls performed in order to compute foo2(**9**) .

   9 (returns 57 (= 1+2+3+…+8+9+12)
   |
   4 (returns 12 (= 1+2+3+4+2)
   |
   2 (returns 2)

g) Compute the result value returned by foo2(**9**).
   57

h) Re-implement this function with memoization (i.e. use a solution array to look-up and store results of recursive calls).
   a. Write the memorized recursive function.

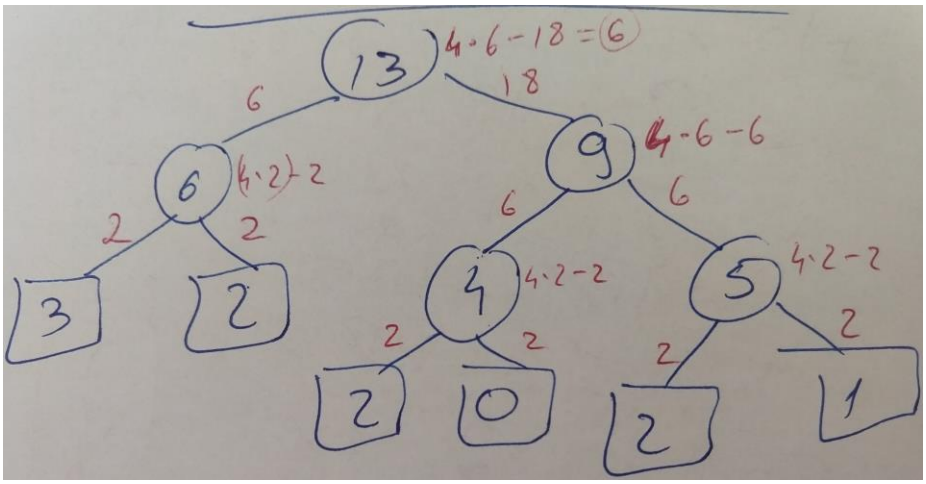
b. In addition to showing the changes in the foo function, **also write the wrapper function** that calls the memorized **foo** function.

P7. Consider this recursive function **foo(N):**

```
int foo3(int N){
    if (N <= 3) return 2;
    int res = 4*foo3(N/2) - foo3(N-4);
    return res;
}
```

a) (6 points) Write the recurrence formula for the **time complexity** of this function (including the base cases) for N>=0. <u>You do NOT need to solve it</u>.
T(N) = c for all N≤3
T(N) = T(N/2) + T(N-4) + c

b) (5 points) Draw the tree that shows the function calls performed in order to compute foo3(13) (the root will be foo3(**13**) and it will have a child for each recursive call.)



c) (3 points) Compute the result value returned by foo3(**13**). **foo3(13) = 6**

d) (7 points) Write code for **foo_mem**, the **memorized** version of **foo** (use a solution array to look-up and store results of recursive calls).