Note: Do not write on the backs; only front pages are digitized and graded.

1. (20 points) Use the Master's theorem to find the complexity of the following functions (assuming T(1) = 1):

1.
$$T(n) = 4n + 4 \cdot T(\frac{n}{4})$$
1. $T(n) = 4n + 4 \cdot T(\frac{n}{4})$
1. $T(n) = 4n + 4 \cdot T(\frac{n}{4}$

2. (15 points) Determine (write it out clearly) the running time equation in terms of n, which is the number of items in the input list L. Based on that equation, find the running time complexity of the function foo. Hint: Pay attention to the running time of Python slicing.

3. (15 points) Determine (write it out clearly) the running time equation in terms of n, which is the number of items between two indices left and right of the input list L. Based on that equation, find the running time complexity of the function bar. Hint: Pay attention to the running time of Python slicing.

```
def bar(L, left, right):

if left >= right:

return 1

sum = 0

for i in range(left, right+1):

sum = sum + i*i

A = L[left:(Rft + right) //2]

return (A,O, k, (A)-1) + bar (A,O, k, (A)-1)

return (A,O, k, (A)-1)
```

4. (10 points) Rewrite the function bar in the previous problem so that your revised version is faster. (Your revised function must, of course, do the same thing as bar does.) <u>Use the Master's theorem</u> to explain why your revised version is faster.

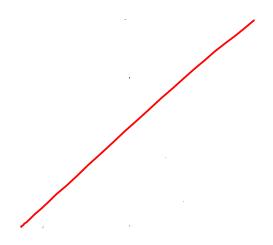
- 7. (20 points) The scenario for this problem is as follows. There are n people and n jobs. Each person has a ranking for the n jobs (smaller rank is better). The goal is to assign each person to each job so that (i) every person has a job and (ii) every job is assigned to exactly one person. The input of this problem is a list of preference list. P[i] is the list of preference of person i. To be consistent the ranking and numbering start from 0 and ends at n-1. Example:
 - 1. P[0] = [1,0,2]. This means person 0 likes job 1 the most, followed by job 0, followed by job 2.
 - 2. P[1] = [2,1,0]. This means person 1 likes job 2 the most, followed by job 1, followed by job 0.
 - 3. P[2] = [0,2,1]. This means person 2 likes job 0 the most, followed by job 2, followed by job 1.

In this example, the best assignment is [1,2,0], which means person 0 is assigned to job 1, person 1 is assigned to job 2, and person 2 is assigned to job 0. The total cost for this assignment is P[0][1]+P[1][2]+P[2][0]=0, which is the best possible (a smaller rank is better implies a smaller cost is better).

Your task is to write a backtracking algorithm to find the best assignment (one with minimal total rank sum). Do that by completing the partially written code below.

```
P = get_preference()  # Assume this function is already defined.
N = len(P)
Solution = [-1]*N  # Hint: this is a permutation problem.
optimal = N*N  # N*N is worse than the worst solution. So it's a safe initial value.

def FindJob(i):
    global optimal
    # Your code goes here
```



FindJob(-1) # This will invoke the backtracking algorithm print("Optimal solution is", optimal)

- 5. (10 points) Consider these three different algorithms:
 - 1. Algorithm A takes as input a list L of n items. It makes two recursive calls on two sublists of L; each sublist has $\frac{n}{2}$ items. Then, Algorithm A combines the result returned by the two recursive calls in n steps.
 - 2. Algorithm B takes as input a list L of n items. It makes four recursive calls on four sublists of L; each sublist has $\frac{n}{2}$ items. Then, Algorithm B combines the result returned by the four recursive calls in only 1 constant step.
 - 3. Algorithm C takes as input a list L of n items. It makes only one recursive call on a sublist of L with $\frac{n}{2}$ items. Then, Algorithm C does additional processing in n^2 steps.

Explain which algorithm gives the fastest running time.

Algorithm A has the fastest aming time. Although B has the benefit of I step, it still has to make H recursive rails and while C only makes A recursive call, it has no steps. Algorithm A has the best scenario for punning time speed, in steps and only 2 recursive calls.

6. (20 points) This problem is known as a bin packing problem. The input is a list of weights w_1, \dots, w_n and a size C. A valid packing is a set of weights that total up to exactly C. For example, given weights 3, 5, 10, 2 and C = 7, the output is True (because 5+2=7). If C = 9, the answer is False (note: no repetition). Your task is to write a backtracking algorithm to print all valid packings. Do that by completing the partially written code below.

Weight = get_weights() # Assume get_weights is already defined. It returns a list of weights.
C = get_capacity() # Assume get_capacity is already defined.
N = len(Weight)
Solution = [-1]*N

def k_pack(i):

Your codes go here

