- # Question 1

We first create a departure array by adding the charging time array to the arrival time array. We then initiate two pointers 1 for arrival array in 1 for departure array. At any point, if the arrival time is greater than the departure time we decrease the number of CP by 1 and move the pointer buy one in departure time and vice versa if the arrival time is smaller than the departure time. We finally return the maximum number of CP At any point for the whole algorithm.
Let arrival time array be A and charging time array be C.

Pseudocode:

```
Dept_array(A, C, n){
        for(i = 0 to n){
                D[i] = A[i] + C[i];
        }
}
Solve(A, D, n){
        sort(A); sort(D);
        i = 0, j = 0, CP = 0, ans = 0;
        while(i < n && j < n) {
                if(A[i] > D[j])     {CP--; j++;}
                else if (arr[i] <= dep[j])    {CP++; i++;}
                ans = max(ans,CP);
        }
        return ans;
}
```

Proof of Correctness:

For the base case with zero cars arriving, the algorithm returns 0 value. Now let's assume the ith iteration to be true, and we need to prove the (i+1)th iteration. we can see that after the ith iteration, the answer variable stores the maximum number of CP required upto ith car arrival. Now for the next iteration, there can be two possible ways either corresponding arrival time will be greater than the departure time or the arrival time will be less than or equal to the corresponding departure time. For the first case the algorithm will reduce the CP by one and increment departure pointer buy one. for the second case the algorithm will increase the CP by one and increment the arrival pointer by one.The maximum of the previous answer and current CP ensures that the algorithm always pics the maximum number of CP required.
Hence proved.

Time & Space Complexity:

We used sorting so time complexity is O(nlogn)
since no extra space is required the space complexity is O(1)

- # Question 2

We need to create a BST and store all the names in that, the BST should be balanced. We then search for the required family name. We can state that we need to print only the subtree of node which we found first. This will ensure the required solution in optimal time.

Pseudocode:

```
Family_Name(string str)        {extracts family name at that node in O(1)}
Middle_Name(string str)        {extracts middle name at that node in O(1)}
First_Name(string str)         {extracts First name at that node in O(1)}
First_Loc_Find(node, name){
        if(node <> NULL)       {return node;}
        a = Family_Name(node → str);
        initialize a variable num which stores -1, 0 or 1 if 2 strings are lexicographically
smaller, equal or greater respectively;
        if(num <> 0)           {return First_Loc_Find(node → left , name);}
        if(num <> 1)           {return node;}
        else                   {return First_Loc_Find(node → left , name);}
}
Print_All(node, name){
        if (node <> NULL)      {return;}
        a = Family_Name(node → str);
        if (a != name)         {return;}
        b = Middle_Name(node → str); c = First_Name(node → str);
        cout << b << " " << c << endl;
        Print_All(node → left, name); Print_All(tree → right, name);
        return;
}
```

Assumptions:

- We need to assume that the strings are small in order to lexicographically compare stings in constant time.
- We need to make our BST balanced in order to perform search operations in O(logn).

Proof of correctness:

While making the BST of all the names we see that all the names with same family name are found in one subtree. So, if we find one family name using Search in BST we can say that all the other names with same family name will be in the following subtree.
We know taht we can do BST Searc in O(log(n)) to find first family name and then in O(k) time we can print all the nodes in that particular subtree.