Problem 1

This problem can be solved using a *Greedy Approach*. At first we find out the time when each particular car departs from the *charging station* using the arrival time array ***arr*** and ***time*** array.

$$departureTime = arrivalTime + timeSpent$$

Now we sort both the arrival and departure time arrays. Then we iterate through these sorted using 2 different iterators*(i,j)*. If the arrival time at the charging station is less than/equal to the departure time, then we need one more charging point and hence we increase the count of variable *charge_point* by 1 and increment the iterator on arrival time array by 1. If the arrival time is greater than departure time, then we will already have one charging point free, so we will reduce *charge_point* by 1 and increment the iterator on departure time array by 1. At each iteration of the loop, we keep updating the max count of charging points in *final_result*, if number of charging points required at each iteration is greater than in *final_result*. In the end we return the value of *final_count*.

Pseudo Code

```
#define ll long long int

void merge(ll *a, ll left, ll right) {

    ll mid = (left + right) / 2;
    ll n = right - left + 1;
    ll temp[n];
    ll i = left; ll j = mid + 1; ll k = 0;

    while ((i <= mid) && (j <= right)) {
        if (a[i] <= a[j]) {
            temp[k] = a[i];
            i++;
        }
        else {
            temp[k] = a[j];
            j++;
        }

        k++;
    }

    while (i <= mid) {
        temp[k] = a[i];
        i++; k++;
    }
    while (j <= right) {
        temp[k] = a[j];
        j++; k++;
    }

    for (ll i = left; i <= right ; i++)
        a[i] = temp[i - left];

}

void mergeSort(ll *a, ll left, ll right) {
    if (left >= right)
        return;
    ll mid = (left + right) / 2;
    mergeSort(a, left, mid);
    mergeSort(a, mid + 1, right);
    merge(a, left, right);

}

chargingPoints(arr, time, n) {

    for (i = 1 : n)
        depart = arr[i] + time[i];

    mergeSort(arr, 0, arr + n);
    mergeSort(depart, 0, depart + n);

    i < - 0; j < - 0;
    charge_point < - 0;
```

```
    final_result <- 0;

    while (i < n and j < n) {
        if (arr[i] <= dep[j]) {
            charge_point++;
            i++;
        }
        else {
            charge_point--;
            j++;
        }

        final_result <- max(charge_point, final_result);
    }
    return final_result;

}
```

Time Complexity

The algorithm makes recursive call to sorting function *mergeSort* and the loop function in *chargingPoints* has complexity cn. So, for the algorithm:

$$\begin{aligned} T(n) &= \underbrace{\mathcal{O}(n \log n)}_{\text{sorting}} + \underbrace{\mathcal{O}(n)}_{\text{looping}} \\ T(n) &= \mathcal{O}(n \log n) \end{aligned}$$

Proof of Correctness

- **Assertion**: At ith iteration let *charge_point* represent number of charging points at station which is equal to number of cars present currently at station
- **Base Case**: Initially charge_point = 0, since no cars are present so no points required.
- At (i-1)th iteration let charging points required to be k, so k cars present.
- If arrival time $<$ departure time, we add 1 to charge_point, so k+1. If arrival time $>$ departure time, we subtract 1 from charge_point, k-1. Hence, we know that at ith iteration *charge_point* represents number of charging points required which is equal to number of cars present at station.

Problem 2

We start by searching for the first instance in BST of the family name given in the input using the function *get_firstInstance*. We print the first name and the middle name of the first instance and subsequently move to left and right child. After getting the first instance of family name, we move to corresponding left and right subtrees and print all the first names and last names using the function *printNames*

Pseudo Code

```
struct nodeBST
{
    string str;
    nodeBST* right;
    nodeBST* left;
};

getFirst_Name(sring str)
getMiddle_Name(string str)
getFamily_Name(string str)

lexCompare(str1, str2) {
    if (str1 == str2)
        return 1;
    else if (str1 > str2)
        return 0;
    else
        return -1;
}

get_firstInstance(nodeBST* root, familyName) {
    if (root == NULL)
```

```
        return root;
    currentName <- getFamily_Name(root->str);
    comp <- lexCompare(currentName, familyName);
    if (comp == 1)
        return root;
    else if (comp == 0)
        return get_firstInstance(root->right, familyName);
    else
        return get_firstInstance(root->left, familyName);
}

printNames(nodeBST* root, familyName) {
    if (root == NULL)
        return root;
    currentName <- getFamily_Name(root->str);
    if (currentName != familyName)
        return;
    firstName <- getFirst_Name(root->str)
    cout << firstName;
    middleName <- getMiddle_Name(root->str)
    cout << middleName;
    printNames(root->left, familyName);
    printNames(root->right, familyName);

    return;
}

solveBSTnames() {
    nodeBST* location = NULL;
    location <- get_firstInstance(nodeBST * root, familyName);
    printNames(location, familyName);

    return;

}
```

Assumptions

- BST is balanced so operations like search take $\mathcal{O}(\log n)$ time.
- All the nodes in the BST with the same family name are connected.
- Since all nodes with the same family name are connected, we may presume that if a node's child has a different name than the parent node, there will be no node in the children subtree with the parent node's family name. Hence, we can reject that entire subtree.
- We make left and right calls in the specified BST based on family name and if the family name is same we consider first and middle name.
- We compare the strings lexicographically, which takes place in constant time.

Time Complexity

We assumed the BST to remain balanced so function *get_firstInstance* takes time $\mathcal{O}(\log_2 n)$ to search. After we get the first instance of the family name, we iterate through the k nodes in the corresponding subtrees which takes $\mathcal{O}(k)$ time.

$$\begin{aligned} T(n) &= \mathcal{O}(\log_2 n) + \mathcal{O}(k) \\ T(n) &= \mathcal{O}(k + \log_2 n) \end{aligned}$$

Proof of Correctness

- At first when we find the first instance of appearance of family name in the BST, we reject the subtree at the node where it is quite sure we won't have the the nodes with the required family name in each step. With this we are sure to have correctly found the first instance of the family name.
- After finding the first instance of family name, only those subtrees at a node get rejected in which we are sure that the required family name will not be found. Hence, we correctly print all the first names and middle names with the given family names.