**Name: Nour Khaled Mahmoud**

**Id:222065**

## Q)6.1.5- Instance simplification

a)

```
function mergeBillsAndChecks(bills, checks) {

    const sb = bills.sort((a, b) => a.telephoneNumber - b.telephoneNumber);
    const sc = checks.sort((a, b) => a.telephoneNumber - b.telephoneNumber);
    let i = 0;
    let j = 0;
    const unpaidTel = [];

 while (i <sb.length && j < sc.length) {
      if (sb[i].telephoneNumber < sc[j].telephoneNumber)
        {
       unpaidTel.push(sb[i].telephoneNumber);
        i++;
     } else if (sb[i].telephoneNumber > sc[j].telephoneNumber) {
        j++;
      } else
       {
        i++;
        j++;
      }
    }
    while (i <sb.length) {
     unpaidTel.push(sb[i].telephoneNumber);
      i++;
    }
    return unpaidTel;
  }
const checks = [
    { telephoneNumber: 310 },
    { telephoneNumber: 110 },
    { telephoneNumber: 420 }
  ];
const bills = [
    { telephoneNumber: 123 },
    { telephoneNumber: 456 },
    { telephoneNumber: 789 },
    { telephoneNumber: 101 }
  ];
```

```
const final = mergeBillsAndChecks(bills, checks);
console.log("The Numbers who failed to pay is:", final);
```

**The time efficiency of this algorithm will be in O (n log n).**

b)

```
function Count(rec) {
    const Counters = new Array(50).fill(0);

    for (let i = 0; i < rec.length; i++) {
      const state = rec[i].state;
      Counters[state]++;
    }
    return Counters;
  }


  const studentRecords = [
    { name: "Nour", state: 1 },
    { name: "Khaled", state: 2 },
    { name: "Mahmoud", state: 3 },
  ];

  const Counters = Count(studentRecords);
  console.log(Counters);
```

**The time efficiency of this algorithm will be in Θ(n)**

## Q)6.3.8- Representation change

False. Because if you have a list [X, Y] for instance and you are searching for the letter Y in the binary search tree it requires 1 comparison while searching for letter Y in the 2-3 tree requires 2 comparisons because the 2 letters will be in the same node.so it’s not the same number of comparisons

## Q)6.6.2- Problem reduction

Multiply each key in the list by -1 and apply a max heap condition algorithm to the new list, then change the signs of all the keys, then the maximum key will be the minimum

## Q)7.2.11- Input Enhancement

**a)** space efficient algorithm

```
function Check(S, T) {
    if (S.length !== T.length) {
      return false;
    }

    const concat = S + S;
    return concat.includes(T);
  }
  const S = "LEAP";
  const T = "PLEA";
  const Final = Check(S, T);

  console.log(Final)
```

**b)** Time efficient algorithm

```
function Check(S, T) {
    if (S.length !== T.length) {
      return false;
    }

    const concat = S + S;
    return concat.indexOf(T) !== -1;
  }

  const S = "LEAP";
  const T = "PLEA";
  const final = Check(S, T);

  console.log(final);
```

## Q)7.3.7- Pre structuring

To ensure distinct elements in a list, begin by initializing an empty hash table. Next, iterate through each element in the list and check its presence in the hash table. If the element exists, it indicates a duplicate, and that means, not all elements are distinct. And, if the element does not exist, insert it into the hash table. Upon completing the iteration, if no duplicates were found, we can say that all elements in the list are distinct. The time efficiency of this approach relies on the list's size and the hash table's implementation efficiency. In general, the average case time complexity for inserting or searching an element in a hash table is O (1). However, in the worst-case scenario of numerous hash collisions, the time complexity can degrade to O(n),

When comparing the efficiency to other algorithms:

1. The brute-force algorithm, which involves comparing each element with every other element, has a time complexity of O(n^2), making hashing a more time-efficient approach for checking distinct elements.
2. The presorting-based algorithm follows a process of sorting the list and then examining adjacent elements for duplicates. By employing efficient sorting algorithms, the list sorting phase operates at a time complexity of O (n log n). and the duplicate check requires iterating through the list once, resulting in a time complexity of O(n). So, the presorting-based algorithm has a time complexity of O (n log n) + O(n), which slower compared to the hashing approach in terms of time efficiency.