Homework 2

Name: Zhaopeng Liu

NetId: zl1732

Problem 1.a

a) master theory

$T(n) = 4T(\frac{n}{3}) + n$

For this recurrence, we have a = 4, b = 3, f(n) = n, and thus we have that $n^{\log_b a} = n^{\log_3 4} = \omega(n^{1.261})$, since $f(n) = O(n^{\log_b a - \epsilon})$, where $\epsilon \approx 0.261$, we can apply case 1 of the master theorem here, the solution is $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 4})$

b) substitution proof

We will guess $T(n) \le cn^{\log_3 4} - 3n$. Then we will have:

$$T(n) \le 4\Big(c(n/3)^{\log_3 4} - n\Big) + n \le cn^{\log_3 4} - 4n + n \le cn^{\log_3 4} - 3n$$

We have the correct answer.

Problem 1.b

a) master theory

$T(n) = 4T(\frac{n}{2}) + n^2$

For this recurrence, we have a = 4, b = 2, $f(n) = n^2$, and thus we have that $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$, since $f(n) = n^2 = \Theta(n^2) = n^{\log_b a}$, we can apply case 2 of the master theorem here, the solution is $T(n) = \Theta(n^{\log_b a} lg(n)) = \Theta(n^{\log_2 4} lg(n)) = \Theta(n^2 lg(n))$

b) substitution proof

We will guess $T(n) \le cn^2 lg(n) + n^2$:

$T(n) \le 4(c(\dfrac{n}{2})^2 lg(n) + (\dfrac{n}{2})^2) + n^2 = cn^2 lg(\dfrac{n}{2}) + 2n^2 = cn^2 lg(n) - cn^2 lg(2) + 2n^2 = cn^2 lg(n) + (2 - cl$

As long as 2-clg(2) < 1, which is $c > \frac{1}{lg(2)}$, we have the correct answer.

Problem 1.c

a) master theory

$T(n) = 4T(\frac{n}{2}) + n^2 lg(n)$

For this recurrence, we have a = 4, b = 2, $f(n) = n^2 lg(n)$, and thus we have that $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$, since the ratio: $\frac{f(n)}{n^{\log_b a}} = \frac{n^2 lg(n)}{n^2} = lg(n)$, f(n) is not polynomially larger, so it falls into the gap between case 2 and case 3, we can't apply the master theorem here

b) substitution proof

We will guess $T(n) \le cn^2 \lg^2 n$:

$$T(n) \le 4T(n/2) + n^2 \lg n \le 4c(n/2)^2 \lg^2(n/2) + n^2 \lg n \le cn^2 \lg(n/2) \lg n - cn^2 \lg(n/2) \lg 2 + n^2 \lg n$$

$T(n) \le cn^2 \lg^2 n - cn^2 \lg n \lg 2 - cn^2 \lg(n/2) + n^2 \lg n \le cn^2 \lg^2 n + (1-c)n^2 \lg n - cn^2 \lg(n/2) \le cn^2$ l
, as long as $c \ge 1$

We have the correct answer.

Problem 2

```
                        T(n)                        -- k * 1
                      /  |   \
                    / ...| ...  \
                  / ...  |  ...  \
            T(n-1)...T(n-1)...T(n-1)                -- k * a
                .        .        .
                .        .        .                 -- k * a^2
                .        .        .                 .
                .        .        .                 .
                .        .        .                 .
     T(1)...  T(1)....T(1)....T(1)...T(1)           -- k * a^(n-1)

  T(0)  T(0)...  T(0)....T(0)....T(0)...T(0)        -- c * a^(n)
```

According to the plot above, $T(n) = k \cdot \sum_{i=0}^{n-1} a^i + c \cdot a^n = k \cdot \frac{a^n - 1}{a-1} + c \cdot a^n$

Problem 3

Iterative Binary Search

In [4]:

```
def iterBinSearch(A, key):
    # list is empty, not valid, return -1
    if len(A) == 0:
        return -1
    # list is not empty
    else:
        lo = 0
        hi = len(A)-1
        # iteration steps
        while lo < hi:
            # update the mid index
            mid = int((lo+hi)/2)
            # find the key
            if A[mid] == key:
                return mid
            # key lies on left side
            elif A[mid] > key:
                hi = mid
            # key lies on right side
            elif A[mid] < key:
                lo = mid + 1
        # not found
        return -1
```

In [5]:

```
A = [1,4,5,3,7,88,6,12,34,9,20,30]
sorted_A = sorted(A)
key = 20
result = iterBinSearch(sorted_A,key)
print("key index: ", result)
print("key == sorted_A[8]?",key == sorted_A[8])
```

```
key index:  8
key == sorted_A[8]? True
```

Loop Invariant:

Initialization:

Before the iteration start, index lo and hi are respectively the first and the last elements of the list, if the target that we search is in the original list, than it must located in A[lo:hi], if the target is not in the original list, then it's also not located in A[lo:hi]

Maintenance:

If target is less than A[mid], than we update the index hi with mid, then target is located in the shorten list A[lo: mid], If target is larger than A[mid], than we update the index lo with mid+1, then target is located in the shorten list A[mid+1, hi]. If target is not in the list, it's neither in A[lo: mid] or A[mid].

Termination:

The iteration terminated when lo == hi or A[mid] == key. In the case "lo==hi", we can think that the list is shorten to length = 1, A[lo:lo(hi)], if the target is in the original list, then it's in A[lo:lo(hi)] and index lo(hi) is the final result, it's correct. if the target is not in the original list, it's not in A[lo:lo(hi)], break out, it's also correct. On the other hand, if iteration terminated with condition A[mid] == key, it means the algorithm find the target and the its index, this is the correct result and the iteration is over.

Runing Time:

- In each iteration, split the list, compare A[mid] and Key and other processes take O(1), so each iteration takes O(1)
- In first iteration, there's n items remains
- In second iteration, there's n/2 items remains
- In k iteration there's n/(2^k) items remains The worst case is the key located at the first or last position in a list. then there will be $\log_2 n$ iterations, and the runnning time is $\Theta(log_2 n)$

Problem 4

Recursive Binary Search

In [6]:

```
def recurBinSearch(A,lo, hi, key):
    # list is empty, invalid case
    if len(A) == 0:
        return -1

    # recursive is not ended
    elif lo < hi:
        result = -1
        mid = int((lo+hi)/2)
        # if find the key, assign the result with its index
        if A[mid] == key:
            result = mid
        # if key lies on left side
        elif A[mid] > key:
            result = recurBinSearch(A,lo, mid, key)
        # if key lies on right side
        elif A[mid] < key:
            result = recurBinSearch(A,mid+1, hi, key)
    return result
```

In [7]:

```
A = [1,4,5,3,7,88,6,12,34,9,20,30]
sorted_A = sorted(A)
key = 20
result = recurBinSearch(sorted_A,0, len(sorted_A)-1, key)
print("key index: ", result)
print("key == sorted_A[8]?",key == sorted_A[8])
```

```
key index:  8
key == sorted_A[8]? True
```

Running Time

It's clear that the running time is $T(n) = T(\frac{n}{2}) + \Theta(1)$

We can see that $n^{\log_b a} = n^{\log_2 1} = \Theta(1) = f(n)$, we can apply case 2 of the master theorem and conclude that the solution $T(n) = \Theta(lg(n))$.

Problem 5

In [8]:

```
class CountInversion(object):
    def findCrossCount(self, B, C):
        i, j = 0, 0
        count = 0
        merged = []
        while i<len(B) and j<len(C):
            next = min(B[i], C[j])
            merged.append(next)

            # next comes from C, then all remaining elements in B is larger than next
            if C[j] == next:
                j += 1
                count += len(B) - i
            # next comes from B
            else:
                i += 1
        # append the remaining elements in either B or C
        merged += B[i:]
        merged += C[j:]
        return count, merged

    def mergeSort(self, A, lo, hi):
        # not valid list case, in which list is empty
        if len(A) == 0:
            return -1

        # base case, return 0 and a list with one elements
        if lo == hi:
            return 0, A[lo:lo+1]
        # recursive steps
        mid = int((lo+hi)/2)
        # count left list, return count and sorted left list
        countLeft, B = self.mergeSort(A, lo, mid)
        # count right list, return count and sorted right list
        countRight, C = self.mergeSort(A, mid+1, hi)
        # count reversion across left and right lists
        countCross, merged = self.findCrossCount(B, C)

        # return the sum of three and final merged sorted list
        return countLeft + countRight + countCross, merged
```

In [9]:

```
s = CountInversion()
A = [1,5,3,2,4,6]
s.mergeSort(A,0,len(A)-1)
```

Out[9]:

```
(4, [1, 2, 3, 4, 5, 6])
```

Runing Time ¶

The running time is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, where $2T(\frac{n}{2})$ is the running time for left and right sublist. $\Theta(n)$ is the running time for findCrossCount function, it taking $\Theta(n)$ to merge two lists with a sum of n elements, and $\Theta(1)$ to update the inversion count.

We can see that $n^{\log_b a} = n^{\log_2 1 2} = \Theta(n) = f(n)$, so we can apply case 2 of the master theorem and conclude that the solution $T(n) = \Theta(n^{\log_b a} lg(n)) = \Theta(nlg(n))$.