Problem set 3

2.3

(a)

$T(n) = 3T(n / 2) + cn$

$= 3 [3T(n/4) + cn/2] + cn = 9T(n/4)+5cn/2$

$= 9[3T(n/8) + cn/4] + 5cn/2 = =27T(n/8)+19cn/4$

$= 27[3T(n/16) + cn/8] + 19cn/4 = 81T(n/16) + 65cn/8$

In general, T(n) can be written as : $T(n) <= 3^kT (n/2^k) + 2cn((3/2)^k - 1)$, and let's substitute k with $\log_2 n$, and we can have:

$T(n) <= n^{\log_2 3}T(1) + 2cn^{0.58}$

Let's assume T(1) = O(1), then we can derive :

$T(n) = O(n^{\log_2 3}O(1)) = O(n^{\log_2 3})$

(b)

$T(n) = T(n-1) + O(1)$, and let's assume $O(1) = c$

$= T(n-1) + c$

$= T(n-2) + c + c = T(n-2) + 2c$

$= T(n-3) + c + 2c = T(n-3) + 3c$

In general, $T(n) <= T(n-k) + kc$, let's say $n = k$, $T(n) <= T(0) + cn$, and let's assume T(0) = a constant.

$T(n) = O(n)$

Therefore, $T(n) = O(n)$

2.5

(a)

$T(n) = 2T(n/3) + 1$

According to the master theorem $T(n) = aT(n/b) + n^d$

$a = 2, b = 3, d = 0$. Since $d < \log_3 2$, the running time complexity is $O(n\log_3 2)$

(b)

$T(n) = 2T(n/3) + 1$

According to the master theorem $T(n) = aT(n/b) + n^d$

$a = 2, b = 3, d = 0$. Since $d < \log_3 2$, the running time complexity is $O(n\log_3 2)$

(c)

$T(n) = 7T(n/7) + n$

According to the master theorem $T(n) = aT(n/b) + n^d$

a = 7, b = 7, d = 1. Since d = $\log_b a$, the time complexity is T(n) = (n log n)

(d)

$T(n) = 9T(n/3) + n^2$

According to the master theorem $T(n) = aT(n/b) + n^d$

In this case, a = 9, b = 3, d = 2

Since d = $\log_b a$, the time complexity is $T(n) = n^2 \log n$

(e)

$T(n) = 8T(n/2) + n^3$

According to the master theorem $T(n) = aT(n/b) + n^d$

In this case, a = 8, b = 2, d = 3.

Since d = $\log_b a$, the time complexity is $T(n) = n^3 \log n$

(f)

$T(n) = 49T(n/25) + n^{3\log n/2}$

According to the master theorem $T(n) = aT(n/b) + n^d$

In this case, a = 49, b = 25, d = 1.5logn

Since d > $\log_b a$, the time complexity if $T(n) = O(n^d) = O(n^{3\log n/2})$

(g)

T(n) = T(n - 1) + 2.

In this case, the pattern of the relation doesn't comply with the standard form of the master theorem, so we can not apply the master theorem.

Instead, we can derive the general term for the T(n)

T(n) = T(n-1)+2

= T(n-2) + 2 + 2 = T(n-2) + 4

= T(n-3) + 2 + 4 = T(n-3) + 6

= T(n-4) + 2 + 6 = T(n-4) + 8

In general, T(n) can be written as:

T(n) = T(n-k) + 2k

And we assume n = k, given T(0) = 1

Then we can haveL

T(n) = T(0) + 2n = O(n)

Therefore, T(n) = O(n)

(h)

$T(n) = T(n-1) + n^c$

In this case, the pattern of the relation doesn't comply with the standard form of the master theorem, so we can not apply the master theorem.

Instead, we can derive the general term for the T(n).

$T(n) = T(n-1) + n^c$

$= T(n-2) + n^c + n^c = T(n-2) + 2n^c$

$= T(n-3) + n^c + 2n^c = T(n-3) + 3n^c$

In general, T(n) can be written as:

$T(n) = T(n-k) + kn^c$. And we assume that n = k, as well as T(0) = 1
$T(n) = T(0) + n*n^c = n^{(c+1)} = O(n^{(c+1)})$

(i)

$T(n) = T(n-1) + c^n$
$= T(n-2) + c^n + c^n = T(n-2) + 2c^n$
$= T(n-3) + c^n + 2c^n = T(n-3) + 3c^n$
In general, $T(n) = T(n-k) + k\,c^n$.
Let's say n = k, given T(0) = 1
$T(n) = T(0) + n*c^n = O(n*c^n) = O(c^n)$

(j)

$T(n) = 2T(n-1) + 1$
$= 2[2T(n-2) + 1] + 1 = 4T(n-2) + 3$
$= 4[2T(n-3) + 1] + 3 = 8T(n-3) + 7$
$= 8[2T(n-4) + 1] + 7 = 16T(n-4) + 15$
In general, $T(n) = 2^k T(n-k) + 2^k - 1$
Substitute n = k, given T(0) = 1
$T(n) = 2^n T(0) + 2^n - 1 = 2 * 2^n - 1$
$= O(2^{(n+1)})$

(k)

$T(n) = T(n^{1/2}) + 1$
$= T(n^{1/4}) + 1 + 1 = T(n^{1/4}) + 2$
$= T(n^{1/8}) + 1 + 2 = T(n^{1/8}) + 3$
In general, $T(n) = T(n^{1/2k}) + k$
And we assume that k = loglogn, given T(1) = 1
Then we can have:
$T(n) = T(1) + loglogn = O(loglogn)$
Therefore, the time complexity of T(n) is O(loglogn)

2.22

Here is the pseudo code for the algorithm:
In this function, we are assuming that the lengths of both arra1 and array2 are larger than 0.

```
kthElement(arr1, arr2, n, m, k)
If n > m
return kthElement(arr2,arr1,m,n,k)

int low = max(0, k-m),  high = min(k,n)

while low <= high

int mid1 = (low+high)/2
```

int mid2 = k - mid1

if mid1 == 0, int l1 = INT_MIN, else l1 = arr1[mid1 - 1]
if mid2 == 0, int l2 = INT_MIN, else l2 = arr2[mid2 -1]
if mid1 == n , int r1 = INT_MAX, else r1 = arr1[mid1]
if mid2 == m, int r2= INT_MAX, else r2 = arr2[mid2]

if l1<=r2 and l2<=r1 return max(l1,l2)
else if l1>r2 high = mid1 -1
else low = mid1 + 1

Explanation:
Assume we have two sub-arrays
array1 : 7, 12, 14, 15
array2: 1, 2, 3, 4, 9, 11

In order to find the kth smallest element, we have to pick some elements from each array.
For example:
7,12| 14, 15
1, 2, 3 | 4, 9, 11

Let's say we are finding the 5^{th} smallest element in the 2 sorted arrays, and we picked 2 elements from the first array, and we picked 3 elements from the second array. Since we are finding the 5^{th} smallest element in the 2 sorted arrays, we have to make sure that 7,12,1,2,3 are the 5 smallest element in the whole number set.

First, it's 100 percent true that 12 is smaller than 14, since they come from the same sorted array, and similarly for 3<4.
Then we have to make sure that 12 is less than 4 and 3 is less than 14 at the same time. In the current case, obviously, 12 is not less than 4, so we have to move the first partition leftward and move the second partition rightward in order that the cross comparison meets our requirement. After shifting, the partition goes like:
7|12,14,15
1,2,3,4|9,11
After movement, we can see that 7 is less than 9, and 4 is less than 12, which means that the left partition is smaller than the right partition. Then, we only have to get the max element from the last element of left array1 and the left array2. And it is max(l1, l2) = 7. Finally, we obtain the 5^{th} smallest element in the 2 sorted arrays.
For the simplicity of the algorithm, each time when we want to shift the array, we are doing the shifting operation on the smaller array and the other index of the partition is just k - the partition index in the first array.
So first, we have to make sure that we are doing the shifting operation on the smaller array. Therefore, we have the recursive call :
if n > m

return kthElement(arr2,arr1,m,n,k).
Next, we are using binary seach to move the partition of each array and compare l1 with r2 each time.
If l1 is larger than r2, then we have to shift the array leftward, otherwise we will have to shift the array rightward until l1<=r2 and l2<=r1.