Homework 5

5.1 1a. Algorithm LargestElement(A[n])

```
right ← n-1
left ← 0
if left = right
    return left
else
    mid ← ⌊(left+right)/2⌋
    pos1 ← LargestElement(A[left...mid])
    pos2 ← LargestElement(A[mid+1...right])
    if A[pos1] ≥ A[pos2]
        return pos1
    else
        return pos2
```

b. This algorithm outputs the position of the leftmost largest element for arrays with several elements of the largest value.

c. $C(n) = C(\lceil \frac{n}{2} \rceil) + C(\lfloor \frac{n}{2} \rfloor) + 1$ for $n > 1$ $C(1) = 0$

$$
\begin{aligned}
C(2^k) &= C(\lceil \tfrac{2^k}{2} \rceil) + C(\lfloor \tfrac{2^k}{2} \rfloor) + 1 \\
&= C(\lceil 2^{k-1} \rceil) + C(\lfloor 2^{k-1} \rfloor) + 1 \\
&= 2C(2^{k-1}) + 1 \\
&= 2(2C(2^{k-2}) + 1) + 1 \\
&= 2^2 C(2^{k-2}) + 2^1 + 1 \\
&= 2^2(2C(2^{k-3}) + 1) + 2 + 1 \\
&= 2^3 C(2^{k-3}) + 2^2 + 2 + 1 \\
&= 2^i C(2^{k-i}) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= 2^k C(2^{k-k}) + 2^{k-1} + 2^{k-2} + \ldots + 1 \\
&= 2^{k-1} \\
&= \boxed{n-1}
\end{aligned}
$$

d. Compared to the brute-force algorithm, this algorithm makes the same number of key comparisons. But in the brute force algorithm there is no overhead with recursive calls.

5a. $T(n) = 4T(n/2) + n \quad T(1) = 1$

$a = 4 \quad b = 2 \quad d = 1$

$4 < 2^1 \qquad 4 = 2^1 \qquad 4 > 2^1$

$\triangle \quad 4 \not< 2^1 \qquad \times \; 4 = 2^1 \qquad \surd \; 4 > 2^1$

Thus $T(n) \in \Theta(n^{\log_b a})$

$\in \Theta(n^{\log_2 4})$

$\in \Theta(n^2)$

b. $T(n) = 4T(n/2) + n^2 \quad T(1) = 1$

$a = 4 \quad b = 2 \quad d = 2$

$4 < 2^2 \qquad 4 = 2^2$

$\times \; 4 \not< 4 \qquad \surd \; 4 = 4$

Thus $T(n) \in \Theta(n^d \log n)$

$\in \Theta(n^2 \log n)$

c. $T(n) = 4T(n/2) + n^3 \quad T(1) = 1$

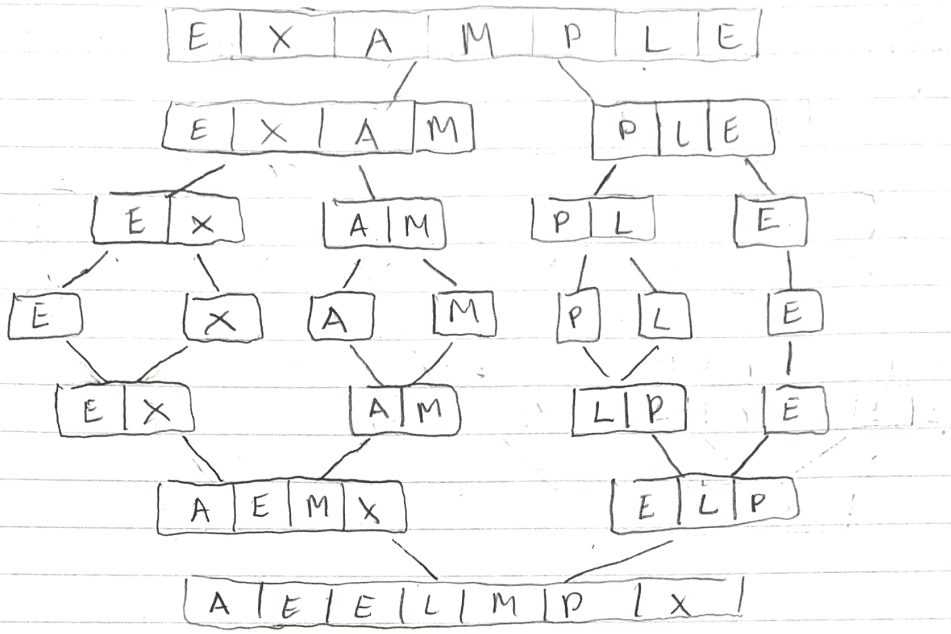
$a = 4 \quad b = 2 \quad d = 3$

$4 < 2^3$

$\surd \; 4 < 8$

Thus $T(n) \in \Theta(n^3)$

6



8a. $C_{worst}(n) = 2C_{worst}(n/2) + n - 1$ for $n > 1$ $\quad C_{worst}(1) = 0$

$$
\begin{aligned}
C_{worst}(2^k) &= 2C_{worst}(2^k/2) + 2^k - 1 \\
&= 2C_{worst}(2^{k-1}) + 2^k - 1 \\
&= 2(2C_{worst}(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1 \\
&= 2^2 C_{worst}(2^{k-2}) + 2^k - 2 + 2^k - 1 \\
&= 2^2(2C_{worst}(2^{k-3}) + 2^{k-2} - 1) + 2^k - 2 + 2^k - 1 \\
&= 2^3 C_{worst}(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1 \\
&= 2^3 C_{worst}(2^{k-3}) + 3(2^k) - 2^2 - 2 - 1 \\
&= 2^i C_{worst}(2^{k-i}) + i(2^k) - 2^{i-1} - 2^{i-2} - \ldots - 1 \\
&= 2^k (worst(2^{k-k}) + k(2^k) - 2^{k-1} - 2^{k-2} - \ldots - 1 \\
&= k2^k - (2^k - 1) \\
&= n\log n - (n+1)
\end{aligned}
$$

b. $C_{best}(n) = 2C_{best}(n/2) + n/2$ for $n > 1$, $C_{best}(1) = 0$

$$
\begin{aligned}
C_{best}(2^k) &= 2C_{best}(2^k/2) + 2^k/2 \\
&= 2C_{best}(2^{k-1}) + 2^{k-1} \\
&= 2(2C_{best}(2^{k-2}) + 2^{k-2}) + 2^{k-1} \\
&= 2^2 C_{best}(2^{k-2}) + 2^{k-1} + 2^{k-1} \\
&= 2^2(2C_{best}(2^{k-3}) + 2^{k-3}) + 2^{k-1} + 2^{k-1} \\
&= 2^3 C_{best}(2^{k-3}) + 2^{k-1} + 2^{k-1} + 2^{k-1} \\
&= 2^i C_{best}(2^{k-i}) + i2^{k-1} \\
&= 2^k C_{best}(2^{k-k}) + k2^{k-1} \\
&= \frac{1}{2} n \log n
\end{aligned}
$$

c. $T_{worst}(n) = 2T_{worst}(n/2) + 2n$

$a = 2 \quad b = 2 \quad d = 1$

$2 \leq 2^1$ ✗ $2 < 2$ $\quad$ $2 = 2^1$ ✓ $2 = 2$

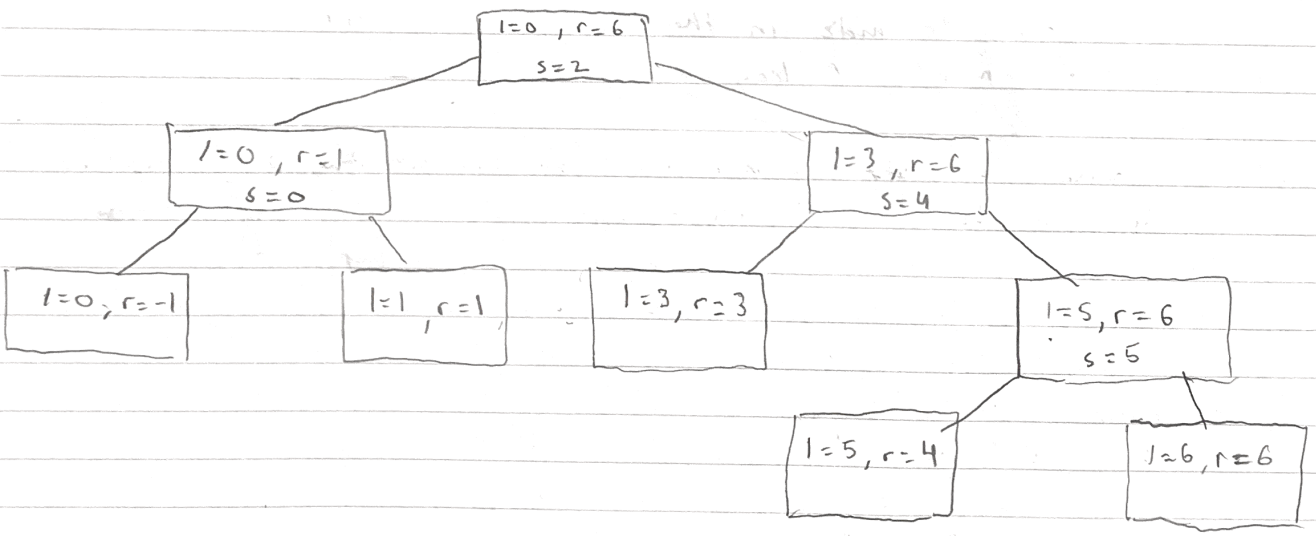
Thus $T(n) \in \Theta(n^d \log n)$
$\in \Theta(n^1 \log n)$
$\in \Theta(n \log n)$

Taking the number of key moves into account does not change the algorithm's efficiency class.

5.2 1.

0	1	2	3	4	5	6
E (p)	X (i)	A	M	P	L	E
E (p)	E	A (i)	M (j)	P	L	X
E	E	A (j)	M (i)	P	L	X
A (p)	E (j)					
A (p, j)	E (i)					
A (p)	E					
E						

0	1	2	3	4	5	6
			M_p	p_i	L	X_j
			M_p	p_i	L_j	X
			M_p	L_i	p_j	X
			M_p	L_j	p_i	X
			L	M_p	p	X
			L		p_p	X_{ij}
					p_{pj}	X_i
					p_p	X
						X



4. With an array with n elements 

and the pivot being the left-most element, the left-to-right scan will get out of bounds if and only if pivot is larger than all the other elements. Appending a sentinal of value equal A[0] after the array's last element will stop will stop the index i of the left-to-right scan of A[0...n-1] from going beyond position n.

A single sentinel will suffice because in quicksort when Partition(A[l...r]) is called for r<n-1, all elements to the right of partition r are greater than or equal to all elements in (A[l...r]), thus A[r+1] will automatically play role of a sentinal to stop index I from going beyond position r+1

5a. For quicksort, an array made up of all equal elements is the best case input. This is because the partitions of array can be made in the middle, thus taking less time and less number of key comparisons.

b. For quicksort, strictly decreasing arrays is the worst case input because if the pivot is the first element, the partition will creat one subarray of size 1 and one subarray of size n-1. This will lead to the recursion depth of n and total time n^2.

9a. Algorithm DutchFlag(A[0...n-1])
// Input: Array of chars R, W, B not in order
// Output: Array of chars in order of R, W, B
i ← 0
j ← 0
k ← n-1
while i<n && A[i]='R' do
i++
while k>=0 && A[k]='B'
k--

```
j = i
while j ≤ k do
    if A[j] = 'R'
        swap (A[i], A[j])
        i++
        j++
    else if A[j] = 'W'
        j++
    else
        swap (A[j], A[k])
        k--
```

b. A solution to the Dutch national flag problem can be used in quicksort, because the array can be divided into three parts arr[l..i] elements are less than pivot, arr[i+1..j-1] elements are equal to pivot and arr[j...r] elements are greater than pivot

5.4 2. $2101 * 1130$

$A = (21 \times 10^2 + 1)$ $B = (11 \times 10^2 + 30)$

$(21 * 11) * 10^4 + (21 * 30 + 1 * 11) * 10^2 + 1 * 30$

$231 * 10^4 + (630 + 11) * 10^2 + 30$

$2310000 + 64100 + 30$

$2,374,130$

6.1 3a. Algorithm LargestSmallest (A[0..n-1]).

// Input array of n≥1 elements
// Output Largest and smallest values in array
for i←0 to n-2 do
    for j←0 to n-2-i do
        if (A[j+1] < A[j] do
            swap (A[j], A[j+1])
largest ← A[n-1]
smallest ← A[0]
return largest, smallest

The efficiency time class of the entire algorithm is $O(n^2) + \Theta(1) + \Theta(1)$ which equals $O(n^2)$.

b. The brute force algorithm's time efficiency would be $\Theta(n)$ since we can find smallest and largest element in array by scanning entire array one time only. The divide and conquer algorithms' time efficiency is also linear since it scans the entire array one time only also. The presorting algorithm above is $\Theta(n^2)$ but using mergesort or quick sort it can be $\Theta(n \log n)$. But brute and divide and conquer algorithm are thus superior to presorting algorithms.

4 $n\log_2 n + k\log_2 n \leq kn/2 \rightarrow k \geq \frac{n\log_2 n}{n/2 - \log_2 n}$

$$k \geq \frac{10^3 \log_2 10^3}{10^3/2 - \log_2 10^3}$$

$$\geq \frac{10^3\left[\frac{3}{\log_3 10}\right]}{\frac{10^3}{2} - \frac{3}{\log_2 10}}$$

$$\geq \frac{10^3 (9.96578)}{500 - 9.96578}$$

$$\geq \frac{9965.78}{490.034}$$

$$= 20.336$$

Thus $k_{min} = 21$ searches needed to justify time spent on presorting an array of 10^3 elements

$$k \geq \frac{10^6 \log_2 10^6}{10^6/2 - \log_2 10^6}$$

$$\geq \frac{10^6 \left[\frac{6}{\log_2 10}\right]}{\frac{10^6}{2} - \frac{6}{\log_2 10}}$$

$$\geq \frac{10^6 [19,9315]}{500000 - (19,9315)}$$

$$\geq \frac{19931500}{499990,0342}$$

$$\geq 39.8637$$

$k_{min} = 40$ searches needed to justify presort of 10^6 elements

7. Algorithm Problem7 (A[0..n-1], s) . .)

// Input : One array, Array A contains real number, and one integer S
// Output : returns 1 if two real numbers sum to S, else returns 0

step 1 : sort the array in ascending order. Mergesort and Quicksort will complish this the Fastest

step2:

```
l ← 0
r ← n-1
while l < r
      if  A[l] + A[r] = s
          return 1
      else if  A[l] + A[r] < s
          l++
      else
          r--
return 0
```

Extra Credit

5.4 6. $m_1 + m_4 - m_5 + m_7 = (a_{00}+a_{11})(b_{00}+b_{11}) + a_{11}(b_{10}-b_{00}) - (a_{00}+a_{01})b_{11} + (a_{01}-a_{11})(b_{10}+b_{11})$

$= a_{00}b_{00} + a_{11}b_{00} + a_{00}b_{11} + a_{11}b_{11} + a_{11}b_{10} - a_{11}b_{00} - a_{01}b_{11} + a_{01}b_{10} - a_{11}b_{10} + a_{01}b_{11} - a_{11}b_{11}$

$= a_{00}b_{00} + a_{01}b_{10}$

$m_3 + m_5 = a_{00}(b_{01}-b_{11}) + (a_{00}+a_{01})b_{11}$

$= a_{00}b_{01} - a_{00}b_{11} + a_{00}b_{11} + a_{01}b_{11}$

$= a_{00}b_{01} + a_{01}b_{11}$

$m_2 + m_4 = (a_{10}+a_{11})b_{00} + a_{11}(b_{10}-b_{00})$

$= a_{10}b_{00} + a_{11}b_{00} + a_{11}b_{10} - a_{11}b_{00}$

$= a_{10}b_{00} + a_{11}b_{10}$

$m_1 + m_3 - m_2 + m_6 = (a_{00}+a_{11})(b_{00}+b_{11}) + a_{00}(b_{01}-b_{11}) - (a_{10}+a_{11})b_{00} + (a_{10}-a_{00})(b_{00}+b_{01})$

$= a_{00}b_{00} + a_{11}b_{00} + a_{00}b_{11} + a_{11}b_{11} + a_{00}b_{01} - a_{00}b_{11} - a_{10}b_{00} - a_{11}b_{00} - a_{00}b_{00} + a_{10}b_{01} - a_{00}b_{01}$

$= a_{10}b_{01} + a_{11}b_{11}$

8. $A(n) = 7A(n/2) + 18(n/2)^2$ for $n > 1$ $A(1) = 0$

$A(2^k) = 7A(2^k/2) + 18(2^k/2)^2$

$= 7A(2^{k-1}) + 18(2^{k-1})^2$

$= 7A(2^{k-1}) + \frac{9}{2}4^k$

$= 7[7A(2^{k-2}) + \frac{9}{2}4^{k-1}] + \frac{9}{2}4^k$

$= 7^2[7A(2^{k-3}) + \frac{9}{2}4^{k-2}] + 7\frac{9}{2}4^{k-1} + \frac{9}{2}4^k$

$= 7^3 A(2^{k-3}) + 7^2\frac{9}{2}4^{k-2} + 7\frac{9}{2}4^{k-1} + \frac{9}{2}4^k$

$= 7^k A(2^{k-k}) + \frac{9}{2}\sum_{i=0}^{k-1} 7^i 4^{k-i}$

$= \frac{9}{2}4^k \sum_{i=0}^{k-1} (\frac{7}{4})^i$

$= \frac{9}{2}4^k \frac{(7/4)^k - 1}{(7/4) - 1}$

$= 6(7^k - 4^k)$

$= 6(7^{\log_2 n} - 4^{\log_2 n})$

$= 6(n^{\log_2 7} - n^2)$