Homework 3

1.

We search for the substring "ABX" in the string "ABCDABCDABCD-ABXD" Successful matches are in bold and unsuccessful are in regular text.

A	B	C	D	A	B	C	D	A	B	C	D	A	B	X	D
A	**B**	X													
	A														
		A													
			A												
				A	**B**	X									
					A										
						A									
							A								
								A	**B**	X					
									A						
										A					
											A				
												A	**B**	**X**	

We see that there are 9 successful and 12 unsuccessful matches.

2. Section 3.2 #8 (a)

ALGORITHM $CountSubstrings(T[0 \dots n])$

$count \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  if $T[i] =$ 'A' **then**
    for $j \leftarrow i$ **to** n **do**
      if $T[j] =$ 'B' **then** $count \leftarrow count + 1$
      end if
    end for
  end if
end for

In the best case, we have a string with no A's in it, and we never enter the second for loop. Thus, we would only iterate through the string once, so

$$C_B(n) \in \Theta(n)$$

In the worst case, we would have a string of just A's, so we would enter the second loop every single time. Thus, we have

$$\begin{aligned} C_W(n) &= \sum_{i=0}^{n-1}\sum_{j=i}^{n} 1 \\ &= \sum_{i=0}^{n-1}(n-i+1) \\ &= (n+1)+(n+0)+\cdots+1 \\ &= \frac{(n+2)(n+1)}{2} \end{aligned}$$

So,

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

3. Section 3.4 #8

If we have an array of n elements, then we can generate a permutation of the array and then check if that permutation is ordered. WE will always have to make $n-1$ comparisons, and at worst, we'll have to check $n!$ permutations. Thus, we'll have to make at most $(n-1)n!$ comparisons. So the efficiency of the worst case is

$$C_W(n) \in O((n+1)!)$$

4. Section 4.1 #7

E |	X	A	M	P	L	E'
E	X |	A	M	P	L	E'
A	E	X |	M	P	L	E'
A	E	M	X |	P	L	E'
A	E	M	P	X |	L	E'
A	E	L	M	P	X |	E'
A	E	E'	L	M	P	X

5. Section 4.1 #11 (a)

The array with the largest number of inversion are reverse sorted arrays, since every $A[i]$ would be larger than every subsequent element. Thus, they would have

$$\sum_{i=0}^{n-1}(n-1-i) = (n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2}$$

number of inversions.

The smallest number would be sorted arrays and they would have 0 inversions.

6. Section 4.4 #8 (a, b, c, d)

(a) Decrease by a constant factor

(b)

$$C(n) = C\left(\frac{n}{3}\right) + 2$$

with $C(1) = 1$ and $n = 3^k$

(c)

$$\begin{aligned} C(n) &= C(3^k) \\ &= C(3^{k-2}) + C(3^{k-1}) + 2 + 2 \\ &\vdots \\ &= C(3^{k-k}) + \underbrace{2 + 2 + \cdots + 2}_{k \text{ times}} \\ &= 1 + 2k \end{aligned}$$

Since $n = 3^k$, then $k = \log_3 n$. So $C(n) = 2\log_3 n + 1$.

(d) The efficiency for binary search is $2\log_2 n+1$. Since the base of the log is smaller for binary than ternary search, then ternary is more efficient. However, both algorithms are of the same efficiency class.

7. Section 4.5 #2

We find the median, and since the size of the array is 7 then

$$k = \lceil 7/2 \rceil = 4$$

We proceed with the quickselect algorithm.

0	1	2	3	4	5	6
s	i					
9	12	5	17	20	30	8
	s	i				
9	5	12	17	20	30	8
		s				i
9	5	8	17	20	30	12
8	5	**9**	17	20	30	12

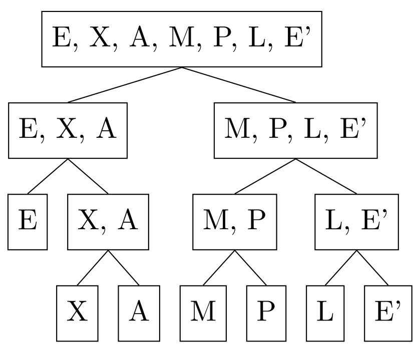
So $s = 2 < 3 = k - 1$, and we partition again with the right subarray.

0	1	2	3	4	5	6
			s	i		
			17	20	30	12
				s		i
			17	12	30	20
			12	**17**	30	20

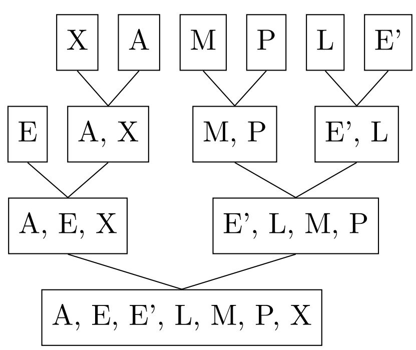
So $s = 4 > 3 = k - 1$, and we partition again with the left subarray. However, the left subarray is a singleton, so no swaps occur. Thus, $s = 3 = k - 1$ and 12 is the median.

8. Section 5.1 #6

For the first half of the tree we have,



Merging back, we get



9. Section 5.2 #5 (a, b)

(a) If the array is made up of all equal elements, then each partition will be exactly in the middle. Thus, this is the best case input.

(b) If the input is a strictly decreasing array, then there will be and empty subarray every time the array is partitioned. Thus, this is the worst case input.

10. Section 5.3 #8 (a)