1 Written Questions

1.1 Q1 a.

Given 5 unique cards, the probability of picking 1 card is $\frac{1}{1}$. The probability of picking a second card that is alphabetical order is $\frac{1}{2}$. It follows that to pick 4 cards in alphabetical order we have $(\frac{1}{1})(\frac{1}{2})(\frac{1}{3})(\frac{1}{4})$ or, more simply put, $\frac{1}{4!} = \frac{1}{24}$.

1.2 Q1 b.

In general, the number of cards in n is irrelevant to the number of permutations of picking k cards in ascending order. It follows from (a) that for picking any number of k cards in ascending order, we have $\frac{1}{k!}$.

2 Q2

Listed from fastest growth to slowest growth:

$f_7(n) = 2^{2^n}$

$f_2(n) = (n + 0.001)!$

$f_6(n) = 2n * 2^n$

$f_9(n) = (\frac{99}{98})^n$.

$f_8(n) = n^{lglg(n)}$

$f_5(n) = lg^2 n$

$f_1(n) = \sqrt{2}^{log_2(n)} = f_3(n) = \sqrt{n}$

$f_4(n) = \sqrt{lg(n)}$

To test these algorithms, I compared each one with large values of n, as well as their graphs.

We know $f_4(n)$ is the slowest algorithm because, in general, $lg(n)$ has the slowest growth rate of any time complexity. Since this value is also square rooted, the time complexity is even slower. Additionally, when compared to all other algorithms for large values of n, $f_4(n)$ grows the slowest.

For $f_1(n)$ and $f_3(n)$ we can use change of base on $f_1(n) = \sqrt{2}^{log_2(n)}$ to get $f_1(n) = n^{log_2(\sqrt{2})}$; $n^{log_2(\sqrt{2})} = 0.5$ and $n^{0.5} = \sqrt{n}$. Thus, $f_1(n) = f_3(n)$.

$f_5(n) > f_1/f_3(n)$ because when graphed, due to the squared exponent, $f_5(n)$ grows significantly faster for large values of n.

$f_8(n) > f_5(n)$ because $f_8(n)$ has an n in the exponent. When graphed, this appears to be the case as n gets very large.

$f_9(n) > f_8(n)$ because although both have an n in the exponent, the exponent for $f_9(n)$ grows much faster and results in a faster growth rate.

$f_6(n) > f_9(n)$ because both have an exponent of n but $\frac{99}{98} < 2$ implies $f_9(n)$ grows slower.

$f_2(n) > f_1(n)$ because, in general, factorials increase much faster than powers of n as n gets very large.

$f_7(n) > f_2(n)$ because an exponent of 2^n grows much faster than a factorial.

3 Q3

3.1 a.

MAXIMUM-SUBARRAY

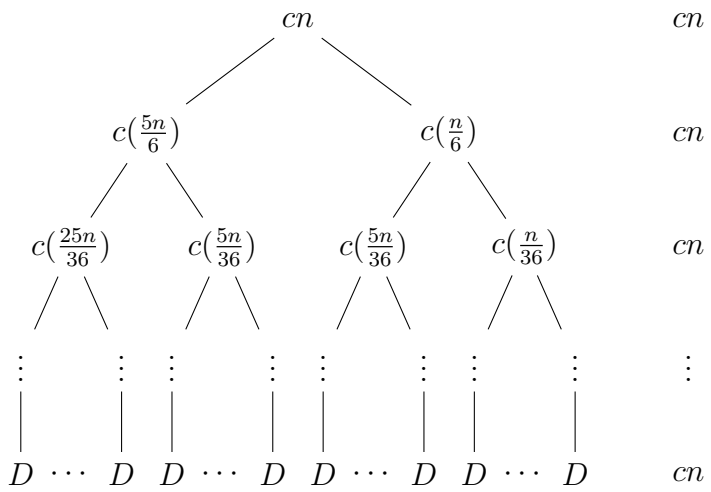
```
max = min[A]
low = 0
high = 0

for i = 0 : A.length
sum = 0

    for j = 0 : A.length
        sum += A[j]
        if (sum > max)
            max = sum
            low = i
            high = j
return(low, high)
```

3.2 b.

In the first loop, we iterate through each value in the array. Additionally, for each index in loop 1, loop 2 iterates through the array from each of its indices to the end of the array. Essentially, we iterate through an array of size n, n times. This implies that our algorithm is $\Theta(n^2)$.

4 Q4

Given the recurrence $T(n) = T(\frac{5n}{6}) + T(\frac{n}{6}) + cn$, where c is a constant, we want to show that the recurrence $T(n)$ is $\Omega(nlgn)$. First, we can draw a recurrence tree:



Each node of this tree is splitting in half which implies that it is dividing n, $lg(n)$ times. Additionally, we can see that at each subsequent level of the tree, n amount of work is being done. Since there is n (times a constant) amount of work being done $lg(n)$ times, our $T(n)$ $= \Omega(nlgn)$.