1. (10 points) For each of the following claims, determine whether they are true or false. Justify your determination (show your work). If the claim is false, state the correct asymptotic relationship as O, Θ, or Ω. Unless otherwise specified, lg is log_2.

(a)	$n+1$	$=$	$O(n^4)$
(b)	2^{2n}	$=$	$O(2^n)$
(c)	2^n	$=$	$\Theta(2^{n+7})$
(d)	1	$=$	$O(1/n)$
(e)	$\ln^2 n$	$=$	$\Theta(\lg^2 n)$
(f)	n^2+2n-4	$=$	$\Omega(n^2)$
(g)	3^{3n}	$=$	$\Theta(9^n)$
(h)	2^{n+1}	$=$	$\Theta(2^{n \lg n})$
(i)	$\sqrt{n}$	$=$	$O(\lg n)$
(j)	10^{100}	$=$	$\Theta(1)$

(a)
TRUE because:
$\lim\ n \to \infty \frac{n+1}{n^4} = (0)$

(b)
FALSE it is not (O) because:
$\lim\ n \to \infty \frac{2^{2n}}{2^n} \neq (0)$

Instead our asymptotic relationship (O) is (Ω) because:
$\lim\ n \to \infty \frac{2^{2n}}{2^n} = (\infty)$

(c)
TRUE because:
$\lim\ n \to \infty \frac{2^n}{2^{n+7}} = \frac{1}{128}$
$\frac{1}{128}$ is a constant, so yes it is (Θ)

(d)
FALSE it is not (O) because:
$\lim\ n \to \infty \frac{1}{1/n} = (\infty)$

This means our asymptotic relationship (O) it is (Ω)

(e)
TRUE because:
$\lim\ n \to \infty \frac{ln^2 n}{lg^2 n} = ln^2(2)$

$ln^2(2)$ is a constant, this means our asymptotic relationship is true.

(f)
FALSE It is not (O) because:
$\lim\ n \to \infty \frac{n^2+2n-4}{n^2} = (1)$

Our original statement of Ω is true but it is also (O) as well, which means the original statement is Θ

(g)
FALSE because:
$\lim\ n \to \infty \frac{3^{3n}}{9^n} = (\infty)$

Its big Ω and its not (O) because its not zero.

(h)
FALSE because:
$\lim\ n \to \infty \frac{2^2 n+1}{2^n lgn} = (0)$

(i)
FALSE because:
$\lim\ n \to \infty \frac{sqrtn}{lgn} = (\infty)$

(j)
TRUE because:
$\lim\ n \to \infty \frac{100^{10}}{1} = constant$
Θ = true

2. (25 points) Asymptotic relations like O, Ω, and Θ represent relationships between *functions*, and these relationships are transitive. That is, if some $f(n) = \Omega(g(n))$, and $g(n) = \Omega(h(n))$, then it is also true that $f(n) = \Omega(h(n))$. This means that we can sort *functions* by their asymptotic growth.[1]

   Sort the following *functions* by order of asymptotic growth such that the final arrangement of functions $g_1, g_2, \ldots, g_{12}$ satisfies the ordering constraint $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{11} = \Omega(g_{12})$.

n	n^2	$(\sqrt{2})^{\lg n}$	$2^{\lg n}$	$n!$	$(\lg n)!$	$\left(\frac{3}{2}\right)^n$	$n^{1/\lg n}$	$n \lg n$	$\lg(n!)$	e^n	42

   Give the final sorted list and identify which pair(s) functions $f(n), g(n)$, if any, are in the same equivalence class, i.e., $f(n) = \Theta(g(n))$.

   (a): Using function g(n):

   $g_1 = n$
   $g_2 = n^2$
   $g_3 = (\sqrt{2})^{lgn}$
   $g_4 = 2^{lgn}$
   $g_5 = n!$
   $g_6 = (lgn)!$
   $g_7 = (\frac{3}{2})^n$
   $g_8 = n^{\frac{1}{lgn}}$
   $g_9 = nlg(n)$
   $g_{10} = lg(n!)$
   $g_{11} = e^n$
   $g_{12} = 42$

[1] The notion of sorting is entirely general: so long as you can define a pairwise comparison operator for a set of objects $\mathcal{S}$ that is transitive, then you can sort the things in $\mathcal{S}$. For instance, for strings, we use a comparison based on lexical ordering to sort them. Furthermore, we can use any sorting algorithm to sort $\mathcal{S}$, by simply changing the comparison operators $>$, $<$, etc. to have a meaning appropriate for $\mathcal{S}$. For instance, using Ω, O, and Θ, you could apply QuickSort or MergeSort to the functions here to obtain the sorted list.

We know the growth of factorial > exponential>function Growth of Polynomial Hence our largest functions are begin with factorial functions.

We know n! > (lg n)! > lg (n!) because lg*n is usually 5 or less as apposed to just n, iterated log functions grow slowly

Hence since our growth of exponential functions > than polynomial functions. We already know our next higher $(g_n)is(e^n)$.

Since we know e is about 2.71. our next highest base to the nth degree would (3/2), which would make this our second highest function.

Our (lg n)! function is next because the factorial will grow faster than n^2 and 2^{lgn} because out $n > lgn$ our factorials out pace the exponents.

After our numerical exponential and factorial functions, we focus on the remaining amount of functions and lgs.

This includes are logarithmic iteration conjoined by because of their equivalence. hence we pair $nlgnandlg(n!)$

we know n lg n + ln(n!) > n

Our remaining functions have set (lgn) to the exponent, logarithmic growth is the inverse of exponential growth and is very slow. hence they are last right before a constant which is going nowhere. No matter what n value.

Please note + represent 'and' below

$n! > e^n > (\frac{3}{2})^n > (lgn)! > n^2 > 2^{lgn} > nlgn+lg(n!) > n+2^{lgn} > (\sqrt{2})^{lgn} > n^{\frac{1}{lgn}} > 42$

$e^n = \Omega(2^n * n)$

lg(n!)=$\Theta(nlgn)$
via 3.19

$n! = \Theta(n^{n+1/2}e^{-n})$
via 3.18

3. (30 points) Professor Dumbledore needs your help optimizing the Hogwarts budget. You'll be given an array A of exchange rates for muggle money and wizard coins, expressed as integers. Your task is to help Dumbledore maximize the payoff by buying at some time i and selling at a future time $j > i$, such that both $A[j] > A[i]$ and the corresponding difference of $A[j] - A[i]$ is as large as possible.

   For example, let $A = [8, 9, 3, 4, 14, 12, 15, 19, 7, 8, 12, 11]$. If we buy one stock at time $i = 2$ (assuming 0 indexing) with $A[i] = 3$ and $j = 7$ with $A[j] = 19$, Hogwarts gets an income of $19 - 3 = 16$ coins.

   (a) Consider the pseudocode below that takes as input an array A of size n:

   ```
   makeWizardMoney(A) :
       maxProfitSoFar = 0
       for i = 0 to length(A)-1 {
           for j = i+1 to length(A) {
               coins = A[j] - A[i]
               if (coins > maxProfitSoFar) { maxProfitSoFar = coins }
       }}
       return maxProfitSoFar
   ```

   What is the running time complexity of the procedure above? Write your answer as a Θ bound in terms of n.

   (b) Explain (1–2 sentences) under what conditions on the contents of A the `makeWizardMoney` algorithm will return 0 gold.

   (c) Professor Dumbledore knows you know that `makeWizardMoney` is wildly inefficient. He suggests you write a function to make a new array M of size n such that

   $$M[i] = \min_{0 \le j \le i} A[j] \ .$$

   That is, $M[i]$ gives the minimum value in the subarray of $A[0..i]$.

   What is the running time complexity of the pseudocode to create the array M? Write your answer as a Θ bound in terms of n.

   (d) Use the template code provided and implement the function described in (3c) to compute the maximum coin difference in time $\Theta(n)$.

   (e) Use the template code provided to determine and compare the runtimes for the functions in 2a and 2d. Explain your findings.

(a):

$\Theta(n^2)$
Because there are 2 loops

(b):

$\Theta(n^2)$
The first condition that will make the algorithm return zero coins if there is only one element in the array because the algorithm locks for two separate values.
The second condition is if the array is in descending order because the algorithm depends on the smallest value occurring before the largest one in the array.

(c):

$\Theta((n)log(n))$

Because: $> n, < n^2$
Hence (n)log(n)

(d):

$19 - 2 = 16$

(e):

```
for(int i=0; i<n, i++){
min=A[0];
    for(int j=0 ; j<i ; j++){
    if A[j]<min)
    min=A[j];
                        }
m[i]=min;
                }
```

4. (15 points) Consider the problem of finding the maximum element in a given array. The input is a sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$. The output is an index i such that $a_i \geq a_1 \geq a_2 \geq ... \geq a_n$.

   (a) Write pseudocode for a simple maximum element search algorithm, which will scan through the input sequence A, and return the index of the last occurrence of the maximum element.

   (b) Using a loop invariant, prove that your algorithm is correct. Be sure that your loop invariant and proof covers the initialization, maintenance, and termination conditions.

(a.)

```
Search(A, v){

for(int i=0 to length(A)){
    if(A[i]==v){
        return i}
        }
    return NIL
}
```

(b):
Loop invariant: A[0] $\leq$ A[n], if the target exists in A.

Initialization: At the start of the iteration of the loop the code checks if A[i]=v. In the best case, it returns i and exits the loop.

Maintenance: It checks all input sequences comparing it to v and returns the sequence location when it finds a match or exits if it doesn't.

Termination: At the termination, it will return i if v is found or NIL if it isn't found. If the loop holds for Initialization, Maintenance and Termination, the it shows that the algorithm is correct.

5. (20 points) Crabbe and Goyle are arguing about binary search. Goyle writes the following pseudocode on the board, which he claims implements a binary search for a target value `v` within an input array `A` containing n elements sorted in ascending order.

```
bSearch(A, v) {
   return binarySearch(A, 1, n-1, v)
}

binarySearch(A, l, r, v) {
   if l <= r then return -1
   m = floor( (l + r)/2 )
   if A[m] == v then return m
   if A[m] > v then
     return binarySearch(A, m+1, r, v)
     else return binarySearch(A, l, m-1, v)
```

   (a) Help Crabbe determine whether this code performs a correct binary search. If it does, prove to Goyle that the algorithm is correct. If it does not, state the bug(s), fix the line(s) of code that are incorrect, and then prove to Goyle that your fixed algorithm is correct.

   (b) Goyle tells Crabbe that binary search is efficient because, at worst, it divides the remaining problem size in half at each step. In response Crabbe claims that fournary search, which would divide the remaining array A into fourths at each step, would be *way more* efficient asymptotically. Explain who is correct and why.

(a):
The code is not correct.it does not perform correct binary search. It looks like from the second line of the algorithm that 1 would automatically be returned. I would change that line of code to be:
If l = r, return -1
Using induction we can prove that the new algorithms correct by showing. that it works for 0 elements, 1 element, and an arbitrary number of elements.

(b):
Goyle is correct because applying binary search, you have log2n+ O(i) where four-ary search has 2 * log4n+O comparisons. Although four-ary search you need to perform three comparisons to cut the search space into four parts, instead of one to cut it into two. so even though four-nary search effectively cut away 3/4 of the search spare at each iteration instead of 2 with binary search, it takes more comparisons to do so,and is less efficient.