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 $\lim_{n\to\infty} \frac{n+1}{n^4} = 0$ So n^4 is a upper bound making this true.

   (b) False $\lim_{n\to\infty} \frac{2^{2n}}{2^n} => \frac{2^n}{1} = \infty$ make 2^n a lower bound not a upper bound .
   (c) True if using $f(n) = 2^n, g(n) = 2^n + 7, c_1 = \frac{1}{2^7}, c_2 = 1, n_0 = 1$ then$\frac{2^{n+7}}{2^7} \leq 2^n \leq 2^{n+7} => \frac{2^7 2^n}{2^7} \leq 2^n \leq 2^{n+7} => 2^n \leq 2^n \leq 2^7 2^n$ so the definition hold for this case.
   (d) False $\lim_{n\to\infty} \frac{1}{1/n} => n = \infty$ because it reaches infinity its a lower bound not upper
   (e) True if $f(n) = \ln^2 n, g(n) = \lg^2 n, c_1 = 0.1, c_2 = 2, n_0 = 1$ it can be seen when reduced to $0.1 \leq \frac{e}{2} \leq 2$ because $ln = log_e n$ so the tight bound definition holds.
   (f) True $\lim_{n\to\infty} \frac{n^2+2n-4}{n^2} => \frac{2n+2}{2n} => \frac{2}{2} = 1$ This means f(n) will always be above if $n_0 >= 2$ so n^2 is always lower than $n^2 + 2n - 4$
   (g) False 9^n should be the lower bound because $\lim_{n\to\infty} \frac{3^{3n}}{9^n} => \frac{3^n 3^n 3^n}{3^n 3^n} => \frac{3^n}{1} = \infty$ so 9^n is the lower bound
   (h) False $\lim_{n\to\infty} \frac{2^{n+1}}{2^{n \ln n}} => \frac{n+!}{\frac{n \ln n}{\ln 1}} => \frac{\ln 2}{\ln n} = 0$ so $2^{n \ln n}$ is the upper bound.
   (i) False $\lim_{n\to\infty} \frac{\sqrt{n}}{\lg n} => \frac{1/2\sqrt{n}}{1/n} => \frac{n}{2\sqrt{n}} => \frac{-1}{4n^{3/2}} = 0$ becuase in approaches 0, $g(n) = \lg n$ is the lower bound not the upper.

(j) True $f(n) = 10^{100}g(n) = 1c_1 = 10^{100}c_2 = 10^{100}n_0 = 1$ so $10^{100} \leq 10^{100} \leq 10^{100}$ so by tight bound definition it holds.

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., $f(n) = \Theta(g(n))$.
when $n_0 = 42^2$
$n! = \Omega(e^n), e^n = \Omega(\left(\frac{3}{2}\right)^n), \left(\frac{3}{2}\right)^n = \Omega(n^2), n^2 = \Omega((\lg n)!), (\lg n)! = \Omega(n \lg n), n \lg n = \Omega(\lg(n!)), \lg(n!) = \Omega(n), n = \Omega(2^{\lg n}), 2^{\lg n} = \Omega(\sqrt{2}^{\lg n}), \sqrt{2}^{\lg n} = \Omega(42), 42 = \Omega(n^{\frac{1}{\lg n}}), n^{\frac{1}{\lg n}} = \Omega(1)$

There are 2 equivelent in class:
1. n and $2^{\lg n}$ as $2^{\lg n}$ simplyflied is n so it can be said they are the same f(n)
2.The constant 42 and $n^{\frac{1}{\lg n}}$ as the limite of $n^{\frac{1}{\lg n}} = 2$ so they can both be consider a constant C and so share the same lower bound and tight bound $\Omega(1)$or $\Theta(1)$

[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.

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.

   the running time complexity for the algorithms is $\Theta(n^2)$

   (b) *Explain (1–2 sentences) under what conditions on the contents of A the* `makeWizardMoney` *algorithm will return 0 gold.*
   The profit would be Zero if A is descending or has the same repeated values. This is because no profit can be made in these 2 cases.

   (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.

```
M(A):
    m=[0]
    x=0
    for i=0 to len(A):
        if A[i]<A[m[x]]
             m.append(i)
             x++
```

the running time complexity of this would just be $\Theta(n)$ as it only iterates through the loop once.

(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.*

The average run time for part A was 0.15 while my average runtime for part B was 0.0006. By looking at the 2 average run times Part D algorithms is a lot more efficient than Part A. I think I made mine a bit more efficient by have it check the elements betweens the elements in subarray [m[i]...m[i+1] instead m[i]....len(A) as the elements after i+1 wouldn't matter because m[i+1]¡m[i].

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.*

```
maximum(A):
     x = 0
     r = A.length-1
     for int=1 to r:
          if A[i] >= A[x]
               x = i
     return i
```

   (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.*

   LI: At the start of every iteration A[x] $\geq$ all the elements in Subarray A[0...i-1].

   Init (i=1): When i = 1 then the subarray is A[0] which is equal to A[x] (A[0]) so LI holds.

   Main(i=k): Assuming our LI hold for i = k-1 then A[x] hold the largest element in subarray A[0,k-1] so when you start the next iteration i=k there are 2 routes. M.1 Suppose A[k]¿A[x], x then gets reassign to i (line 4) so right after the iteration A[x] is $\geq$ then all the elements in subarray A[0..k] otherwise M.2 A[k]¡A[x] so A[x] is still $\geq$ to the elements in subarray A[0....k]. So LI holds in both cases.

   Term (i=r+1): The LI stats that A[x] $\geq$ all the elements in array A[0...i-1] and i=r+1 so its the subarray A[0...R] which is the whole array A. so By LI we proves the A[x] is the maximum element in A and we return i.

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.*

It would but it has coding erros.
line 2 should either be binarySearch(A,1,n,v) or (A,0,n-1,v) depending on the program because the last element wouldn't be included
line 5 should be if l¿r then return -1 otherwise it would always return -1
line 8 should be A[m] ¡ v then otherwise it's flipped around and would go to wrong half
line 10 should not be intended this is more depending on the program
I would also add another line before 5 stating if loop.isEmpty return -1 like this if they try to pass it an empty loop it wouldn't be able to fine v so just returns v

Basecase n=0 or n=1: when n = 0 the array A is empty so doesn't locate v therefore returning -1.

Induction Hypo: binarySearch correctly locates v within inpute array A then returns it's location otherwise returns -1.

Inductive step (n): recursively divides the array by 2 pick the side where v is located until either its in the middle of a subarray or when the subarray only

contains 1 element and comparing that to v and if its equal to v then return m but if it is not then iterates again and returns -1 like stated in IH.

(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.*

I agree the dividing the array A into fouths would be more effecient but I wouldn't say way more efficient. When dividing by 2 you can trivally say it has a tight bound of $\Theta(\log_2(n))$ and then dividing it by 4 would be $\Theta(\log_4(n))$ and when n gets greater there isn't a huge difference as $\lim_{n\to\infty} \frac{\log_4(n)}{\log_2(n)} = 0.5$ because it's a constant and not infinity we can say dividing by forths is more efficient but not way more efficient as you can put the same tight bound of $\Theta(\log(n))$.