1. (1 pt) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in 8n2 steps, while merge sort runs in 64nlgn steps. For what values of n does insertion sort run faster than merge sort?

**Note:** lg n is log “base 2” of n or . There is a review of logarithm definitions on page 56. For most calculators you would use the change of base theorem to numerically calculate lgn.

For this question, that let us to find n when 8n2 <= 64nlogn.

So,

8n2 <= 64nlogn

n2 <= 8nlogn

n <= 8logn

n – 8logn = 0

n = 43.411

So for 0 < n <= 43, insertion sort works better than merge sort.

1. (5 pts) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.
   1. f(n) = n0.25; g(n) = n0.5
   2. f(n) = n; g(n) = log2 n
   3. f(n) = log n; g(n) = ln n
   4. f(n) = 1000n2; g(n) = 0.0002n2 – 1000n
   5. f(n) = nlog n; g(n) =*n*
   6. f(n) = en; g(n) = 3n
   7. f(n) = 2n; g(n) = 2n+1
   8. f(n) = 2n; g(n) =
   9. f(n) = 2n; g(n) = n!
   10. f(n) = lgn; g(n) =

f(n) is O(g(n))

f(n) is Ω(g(n))

f(n) is Θ(g(n))

f(n) is Θ(g(n))

f(n) is Ω(g(n))

f(n) is O(g(n))

f(n) is Θ (g(n))

f(n) is O(g(n))

f(n) is O(g(n))

f(n) is O(g(n))

1. (4 pts) Let f1 and f2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.
2. If *f1*(*n*) *=* O(*g*(*n*)) and *f2*(*n*) *=* O(*g(n*))then *f1*(*n*)*=* Θ(*f2(n*) ).

Because we know f1 (n) = o(g(n)) and f2 (n) = O(g(n)). And the f2 can be f2 = O(f2(n)) and f2 = O(g(n)), and f2(n) = Θ (*f2* (n)), so O(g(n)) =Θ (*f2* (n)). Therefore, f1 = O(g(n))= Θ (f2(n)). So f1(n)= Θ (f2(n) ).

1. If *f1*(*n*) *=* O(*g1*(*n*)) and *f2*(*n*) *=* O(*g2*(*n*))then *f1*(*n*)+ *f2*(*n*)*=* O(max{*g1*(*n*)*, g2*(*n*)} )

Let *f1*(*n*) *=* O(*g1*(*n*)) and *f1*(*n*) *=* O(*g1*(*n*)). This means that there exist constants c1, c2 > 0 such that f1(n) <= c1g1(n) and f2(n) <= c2g2(n) for all n > 0 integers. To prove the claim, we must find some constant c3 that causes f1(n) + f2(n) <= c3(g1(n) + g2(n)) for all n > 0 integers.

So,

f1(n) + f2(n) <= c1g1(n) + c2g2(n)

<= max(c1, c2)g1(n) + max(c1, c2)g2(n)

<= max(c1, c2)(g1(n) + g2(n))

= c3(g1(n) + g2(n))

so f1(n)+ f2(n)= O(max{g1(n), g2(n)})

1. (5 pts) **Merge Sort and Insertion Sort Programs**

Implement merge sort and insertion sort to sort an array/vector of integers. These are very common algorithms and you may modify existing code if you reference it. You may implement the algorithms in the language of your choice, name one program “mergesort” and the other “insertsort”. Your programs should be able to read inputs from a file called “data.txt” where the first value of each line is the number of integers that need to be sorted, followed by the integers.

Example values for data.txt:

4 19 2 5 11

8 1 2 3 4 5 6 1 2

The output will be written to files called “merge.out” and “insert.out”.

For the above example the output would be:

2 5 11 19

1 1 2 2 3 4 5 6

***Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named data.txt.***

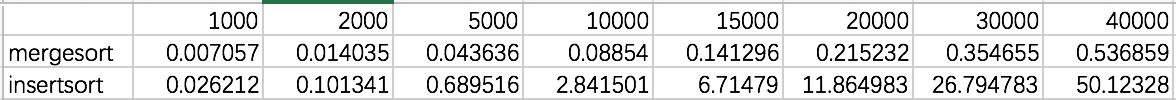
1. (10 pts) **Merge Sort vs Insertion Sort Running time analysis**

The goal of this problem is to compare the experimental running times of the two sorting algorithms.

1. Code

*#!/usr/bin/env python***import** sys  
**import** datetime  
**import** random  
  
**def** mergesort(lst):  
 **if** (len(lst) <= 1): **return** lst  
 left = mergesort(lst[:len(lst) / 2])  
 right = mergesort(lst[len(lst) / 2:len(lst)])  
 result = []  
 **while** len(left) > 0 **and** len(right) > 0:  
 **if** (left[0] > right[0]):  
 result.append(right.pop(0))  
 **else**:  
 result.append(left.pop(0))  
  
 **if** (len(left) > 0):  
 result.extend(mergesort(left))  
 **else**:  
 result.extend(mergesort(right))  
 **return** result  
  
  
*# insert\_sort***def** insertsort(l):  
 **for** i **in** range(len(l)):  
 min\_index = i  
 **for** j **in** range(i + 1, len(l)):  
 **if** l[min\_index] > l[j]:  
 min\_index = j  
 tmp = l[i]  
 l[i] = l[min\_index]  
 l[min\_index] = tmp  
 **return** str(l)  
  
  
**def** main():  
 n = 40000  
 **print** n  
 list = [random.randint(0, 10000) **for** \_ **in** range(n)]  
  
 starttime = datetime.datetime.now()  
 **print** mergesort(list)  
 endtime = datetime.datetime.now()  
 **print** (endtime - starttime)  
  
  
 starttime = datetime.datetime.now()  
 **print** insertsort(list)  
 endtime = datetime.datetime.now()  
 **print** (endtime - starttime)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

1. Table of running time



c)

d)

For merge sort, the curve is more like linear graph, but insertion sort more like polynomial curve.

e)

For merge sort, the theoretical running time is O(nlogn), in my case, the running time is like kn, because when n goes to infinity, the logn will close to a constant value. So when n is big enough, nlogn will becom kn. So, the experimental running time is consistent with theoretical running time.

For insertion sort, the theoretical running time is O(n2), in my case, the running time is like kn2 + cn. Because of the n is infinity, so we just care about n2. So, the experimental running time is consistent with theoretical running time.

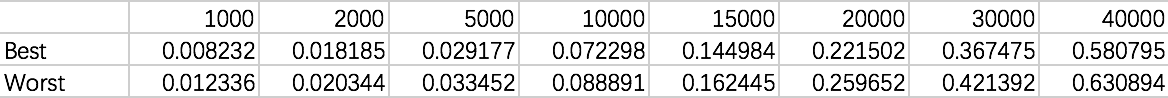
**EXTRA CREDIT:** *It was the best of times, it was the worst of times…*

Generate best case and worst case input for both algorithms and repeat the analysis in parts b) to d) above. Discuss your results.

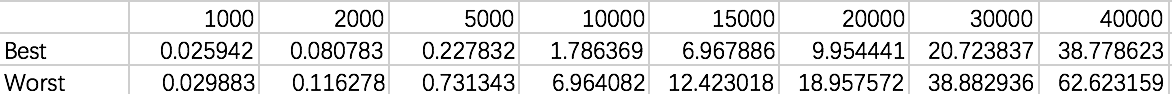
For best case, the array should a sorted array, just like [1,2,3,4,5,6,7,8,9,10,….,n-1,n]

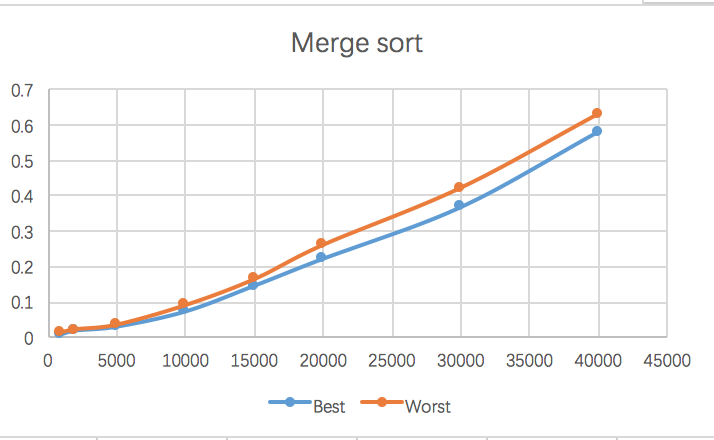
For worst case, the array should a reversed sorted array, just like [n,n-1,….10,9,8,7,6,5,4,3,2,1]

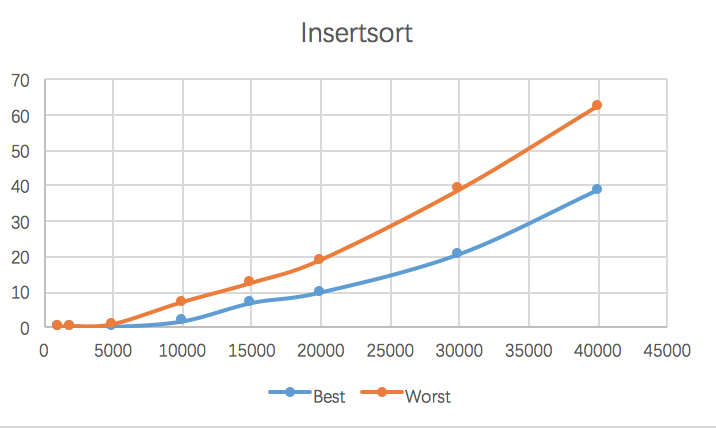
Merge sort



Insertion sort







For merge sort, the two curves are similar, they all like linear graph. For insertion sort, the best case look like linear graph, and worst case like polynomial curve.