CS2223 Algorithms. B Term 2013
Homework 2 Solutions
By Artem Gritsenko, Ahmedul Kabir, Yun Lu, and Prof. Ruiz

Problem 1. **(Solutions by Artem Gritsenko)**

Algorithm:

```
1 for each set Si:
2     for each other set Sj:
3         disjoint = true
4         for each element k of Si:
5             if k belongs to Sj:
6                 disjoint = false
7         if disjoint == true
8             print 'Sets ',i,j,' are disjoint'
9             return(true)
10 return(false)
```

1. Implementation Version 1

DisjointSetsVersion1(L)	**cost**	**times**
1 for i = 0 to len(L)-1	c_1	n+1
2 for j = i+1 to len(L)-1:	c_2	n*([(n-1)/2]+1)
3 disjoint = True	c_3	n*[(n-1)/2]
4 for k =0 to len(L[i])-1:	c_4	n*[(n-1)/2]*(n+1)
5 for s = 0 to len(L[j])-1:	c_5	n*[(n-1)/2]]*n*(n+1)
5 if L[i][k] == L[j][s]:	c_6	n*[(n-1)/2]*n*n
6 disjoint = False	c_7	n*[(n-1)/2]*n*n
7 if disjoint == True	c_8	n*[(n-1)/2]
8 print‘Sets’,i,j,’are disjoint’	c_9	1
9 return(True)	c_{10}	1
10 return(False)	c_{11}	1

The numbers on the lines in the implementation correspond to the numbers on the lines of the Algorithm and mean the same instructions. Now let’s count the total number of instructions executed during the run of the program in the worst case, for each instruction line.

1) The first loop iterates over all the sets to get the 1st set for comparison. The number of executions of the loop condition in the worst case equal to the number of sets plus one. This is because for cycle condition we always have 1 more iteration than for the cycle body. The cycle body, thus would iterate n times.

2) The second loop iterates over all the sets to get the 2nd set for comparison. The second cycle has the number of iterations n/2 because we want to consider only j > i cases. That means we have n-1 iterations for i=0, n-2 iterations for i=1, n-3 iterations for i=2, etc. The total number of iterations would be sum of all numbers from n-1 to 1, which is (n-1)*n/2:

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

As in the previous case we add plus one execution for the condition, though the body of the 2nd loop would iterate (n-1)/2 times.

3) This instruction executes the number of times that both previous loops execute, which is the multiplication of the previous loops number of executions and equal to n*(n-1)/2.

4) The third loop iterated over the elements of the 1st set, which is n iterations in the worst case. As in previous cases we have n+1 executions of the loop condition and n executions of the loop body.

5) The forth loop iterates over the elements of the 2nd set, which is n iterations in the worst case.

5)-7) The loop body is executed a number of times equal to the multiplication of the 4 loops number of iterations.

8)-10) These instructions are executed only once, because if we have found the disjoint sets the program ends. Similarly, if we did not find a pair of disjoint sets, we just return False.

The total number of instructions is the sum of products of corresponding costs and times equal to

T(n) = c_1 * (n+1)
+ c_2 * (n*(((n-1)/2)+1))
+ c_3 *(n*(n-1)/2)
+ c_4 *(n*(n-1)/2*(n+1))
+ c_5 *(n*(n-1)/2*n*(n+1))
+ c_6 *(n*(n-1)/2*n*n)
+ c_7 *(n*(n-1)/2)
+ c_8 *(n*(n-1)/2*n*n)
+ c_9 *1+ c_{10} *1+ c_{11} *1.

We can group and represent it in the way:

T(n) = $k_1 n^4 + k_2 n^3 + k_3 n^2 + k_4 n + k_5$.

2. Claim: $T(n) = O(n^4)$

3. Proof: We need to find n_0 and c that for all $n > n_0$ the following holds: $T(n) \le cg(n)$.

We can note that $k_2n^3 < k_2n^4$, $k_3n^2 < k_3n^4$, $k_4n < k_4n^4$, $k_5 < k_5n^4$ for $n > 1$. That means we can state that T(n) ≤ $k_1n^4 + k_2n^4 + k_3n^4 + k_4n^4 + k_5n^4 = cn^4$ for all $n > 1$, where $c = k_1 + k_2 + k_3 + k_4 + k_5$. Thus, $T(n) = O(n^4)$.

2. Implementation Version 2.

DisjointSetsVersion1(L)	**cost**	**times**
1 for i = 0 to len(L)-1	c_1	n+1
2 for j = i+1 to len(L)-1:	c_2	n*([(n-1)/2]+1)
3 disjoint = True	c_3	n*[(n-1)/2]
4 for k = 0 to len(L[i])-1:	c_4	n*[(n-1)/2]*(n+1)
5 if L[i][k]==1 & L[j][k]==1:	c_5	n*[(n-1)/2]*n
6 disjoint = False	c_6	n*[(n-1)/2]*n
7 if disjoint == True	c_7	n*[(n-1)/2]
8 print 'Sets',i,j,'are disjoint'	c_8	1
9 return(True)	c_9	1
10 return(False)	c_{10}	1

The difference with the implementation1 is that we got rid of one loop, which traversed the elements of the second set for comparison (line 5 in version 1). We can do this because our sets are consistent with each other due to the used data structure. So we can traverse over the elements of only the first set and use the same index for the second set. Thus, the number of instructions for the second implementation would be:

T(n) = c_1*(n + 1)
+ c_2 * n*(((n-1)/2)+1)
+ c_3 * n*(n-1)/2
+c_4 * n*(n-1)/2*(n+1)
+ c_5 * n*(n-1)/2*n
+c_6 * n*(n-1)/2*n
+c_7 * n*(n-1)/2
+c_8 *1 + c_9 *1 + c_{10} *1.

We can group and represent it as follows:
T(n) = $k_1n^3 + k_2n^2 + k_3n + k_4$.

2. Claim: $T(n) = O(n^3)$

3. Proof: We need to find n_0 and c that for all $n > n_0$ the following holds: $T(n) \leq cg(n)$.
We know that $k_2 n^2 < k_2 n^3$; $k_3 n < k_3 n^3$; and $k_4 < k_4 n^3$ for n > 1. That means that $T(n) \leq k_1 n^3 + k_2 n^2 + k_3 n + k_4 = cn^3$, where $c = k_1 + k_2 + k_3 + k_4$. Thus, $T(n) = O(n^3)$.

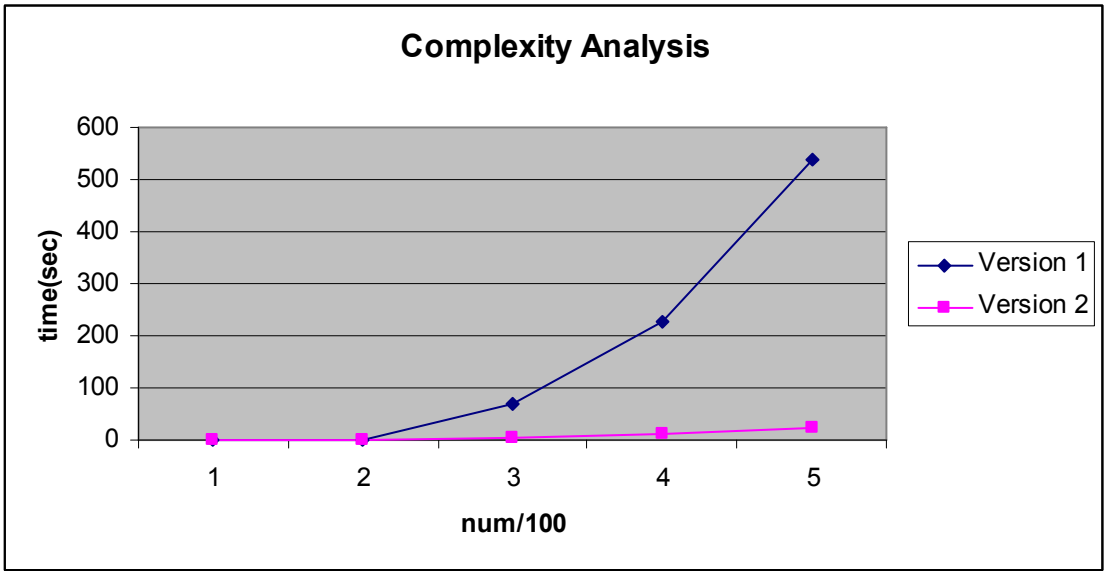
3. Experimental Comparison of the two Versions.

1. See hw2_problem1_code_cs2223_b13.py at:
http://web.cs.wpi.edu/~cs2223/b13/HW/HW2/Solutions/hw2_problem1_code_cs2223_b13.py

2. Execution time.

Input size	Version 1 Runtime (sec.)	Version 2 Runtime (sec.)
100	0.02800107	0.024001837
200	1.732099771	0.18701005
300	70.70604396	4.6963
400	228.4180651	11.38765097
500	539.2678452	22.96531296

3.



The growth of the functions match the asymptotic behavior. The thing that influences the results is the way the sets are randomly generated. If they are sparse, it would be easy (fast) to find a disjoint set. In my implementation that is regulated with a threshold parameter that says with what probability a certain number would be included in the set. The results generated are for the 0.2 probability, which generated not-sparse sets.

Problem 2 (Solutions by Ahmedul Kabir)

Rank the following functions by order of growth. That is, find an arrangement f_1, f_2, ..., f_6 of the functions satisfying $f_1 = \Omega(f_2)$, $f_2 = \Omega(f_3)$, ..., $f_5 = \Omega(f_6)$. Partition your list into equivalent classes such that functions f(n) and g(n) are in the same class if and only if f(n) = Θ(g(n)).

a^n
n^a
a^{n^a}
$\log_a(n)$
a^{a*n}
$n^{\log_a(n)}$

where a is a constant greater than 1. Provide a detailed, rigorous proof of each part of your solution.

Solutions: Below, we'll sort the function in increasing order of growth rate.

Intuitively, $\log_a n$ grows asymptotically slower than the others. So we start by checking whether it is O(n^a)

$$\lim_{n\to\infty} \frac{\log_a n}{n^a} = \lim_{n\to\infty} \frac{\frac{1}{n \ln a}}{an^{a-1}} \quad \text{(using L'Hospital's rule)}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{n \ln a}}{\frac{an^a}{n}} = \lim_{n\to\infty} \frac{1}{an^a \ln a} = 0$$

So, $\log_a n$ = O(n^a)

Now, looking at n^a and a^n, it seems quite obvious that n^a = O(a^n) since the former is polynomial and the latter is exponential. Let us use the limit rule to verify:

$$\lim_{n\to\infty} \frac{n^a}{a^n} = \lim_{n\to\infty} \frac{an^{a-1}}{a^n \ln a} \quad \text{(using L'Hospital's rule)}$$

$$= \frac{a}{\ln a} \lim_{n\to\infty} \frac{n^{a-1}}{a^n} = 0$$

(since for all constants a and b such that a > 1, $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$. See textbook page 55)

Hence, n^a = O(a^n)

We will now use the limits method to find the relationship between a^n and a^{an}.

$$\lim_{n\to\infty} \frac{a^n}{a^{an}} = \lim_{n\to\infty} \frac{a^n \ln a}{a^{an+1} \ln a} \quad \text{(using L'Hospital's rule)}$$

$= \lim_{n\to\infty} \frac{a^n}{a.(a^n)^a} = \lim_{n\to\infty} \frac{1}{a.(a^n)^{a-1}} = 0$ [since the denominator approaches ∞]

So, $\boldsymbol{a^n = O(a^{an})}$**.** For illustration purposes, we include here another proof of the fact $\boldsymbol{a^n = O(a^{an})}$ that uses the definition of Big-Oh directly. We need to show that there exist constants c > 0 and $n_0 > 0$ such that for all $n \geq n_0$, $a^n \leq c\ a^{an}$. Note that since a > 1, $a \leq a^a$. Hence for any $n \geq 1$, $a^n \leq (a^{a)^n} = a^{an}$. Therefore the definition of $\boldsymbol{a^n = O(a^{an})}$ is satisfied with constants c=1 and $n_0 = 1$.

Now for $\boldsymbol{a^{n^a}}$**,** we will compare it with $\boldsymbol{a^{an}}$**.** We take the logarithm of both functions (we'll use $\log_a$). We can do this since log is monotonically increasing and the values of both $\boldsymbol{a^{n^a}}$ and $\boldsymbol{a^{an}}$ are greater than a for large values of n and since a > 1. Taking the log (with base a) on both functions we get $\log_a a^{n^a} = n^a \log_a a = n^a$ which grows faster than $\log_a a^{an} = an \log_a a = an$. Also note that for a > 1, $an \leq n^a$ for any $n \geq 3$. Therefore the definition of $\boldsymbol{a^{an} = O(a^{n^a})}$ is satisfied with constants c=1 and $n_0 = 3$.

Hence $\boldsymbol{a^{an} = O(a^{n^a})}$**.**

So far we have $\boldsymbol{\log_a n < n^a < a^n < a^{an} < a^{n^a}}$ **,** and we need to put $n^{\log_a n}$ in its proper place**.**

Comparing $n^{\log_a n}$ with n^a, note that their values are equal when $a = \log_a n$ or $n = a^a$. But for all values of $n > a^a$, we have $n^a < n^{\log_a n}$. So we have our values $n_0 = a^a$ and c = 1 for which $n^{\log_a n}$ will be asymptotically larger for all $n > n_0$. We can also take the log of both sides and see that $\log_a n^{\log_a n} = (\log_a n)^2$ which grows faster than $\log_a n^a = a \log_a n$ (Since $\log n$ is asymptotically larger than constants).

Hence $\boldsymbol{n^a = O(n^{\log_a n})}$

Now to compare $n^{\log_a n}$ with a^n, we can take the log (with base a) on both sides and see $\log_a n^{\log_a n}) = (\log_a n)^2$ and $\log_a a^n = n \log_a a = n$. We know that $(\log n)^2$ grows slower than n (see a proof of this on page 57 of the textbook).

So $\boldsymbol{n^{\log n} = O(a^n)}$**.**

Our final ordering is therefore $\boldsymbol{\log_a n < n^a < n^{\log n} < a^n < a^{an} < a^{n^a}}$

where f(n) < g(n) is used here as shorthand for f(n) = O(g(n)), or equivalently, g(n) = Ω(f(n)).

Problem 3: (Solutions by Yun Lu)

Find a function f(n) and a function g(n) such that f(n) ≠ O(g(n)), f(n) ≠ Ω(g(n)), and f(n) ≠ Θ(g(n)). Explain your answer in detail.

According to the definition, $f(x) = O(g(x))$, if there exist constants x_0 and c>0 such that $f(x) \leq cg(x)$ for all $x \geq x_0$.

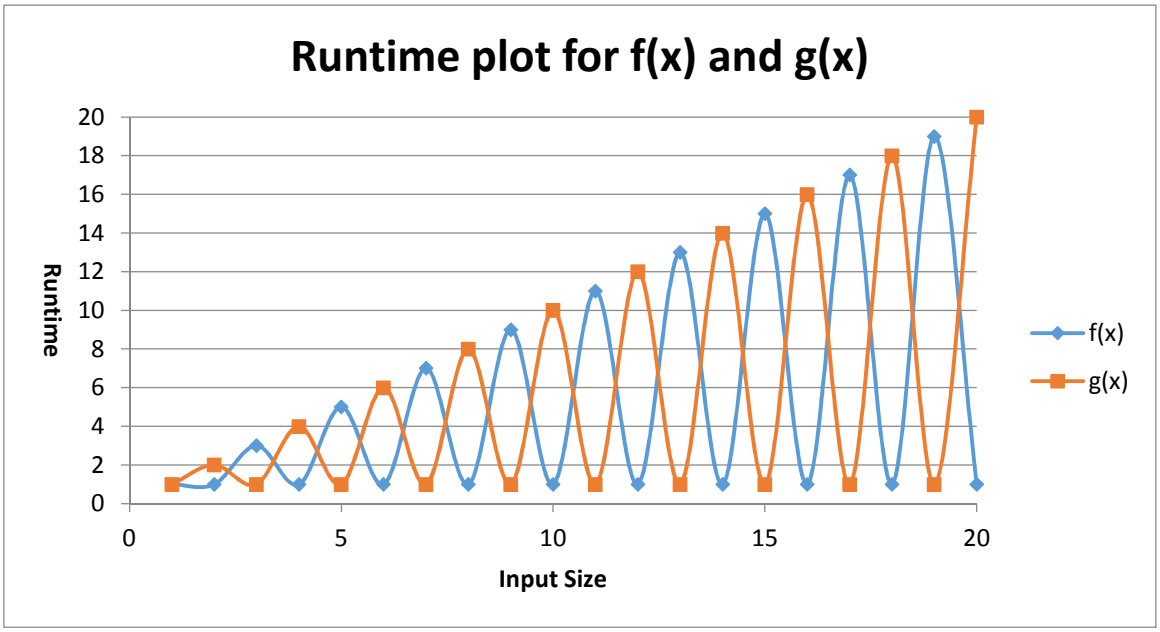
Consider the following functions $f(x)$ and $g(x)$:

$$f(x) = \begin{cases} 1, & x \text{ is even} \\ x, & x \text{ is odd} \end{cases}$$

$$g(x) = \begin{cases} 1, & x \text{ is odd} \\ x, & x \text{ is even} \end{cases}$$

There are no constants c and x_0 to make either $f(x) = O(g(x))$ or $g(x) = O(f(x))$ hold. The reason for this is that no matter how much the input size *x* grows, $f(x) < g(x)$ for even x's, $g(x) < f(x)$ for odd x's, and the difference between $f(x)$ and $g(x)$, $f(x) - g(x) = \pm(x-1)$, keeps increasing alternating from a positive to a negative difference for odd and even x's, as x goes to infinity. Hence, no constants c and x_0 exist that would make either $f(x) \leq cg(x)$ or $g(x) \leq cf(x)$ for all $x \geq x_0$.

Below is the runtime plot for $f(x)$ and $g(x)$:



Note: There many other pairs of functions f(n) and g(n) that satisfy the conditions of this problem, that is that their asymptotic growth are incomparable. So this is not the only possible solution to this problem.

Problem 4: (Solutions by Artem Gritsenko)

Write a formal mathematical proof of the following property of Big-O:
Transitivity of Big-Oh: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

Solution:

To show that Big-Oh is transitive, we must show that for all functions $f(n)$, $g(n)$ and $h(n)$, if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$. Assume that $f(n) = O(g(n))$ and $g(n) = O(h(n))$.

- Since $f(n) = O(g(n))$, there exist constants n_1 and c_1 such that $f(n) \leq c_1 g(n)$ for all $n \geq n_1$
- Since $g(n) = O(h(n))$, there exist constants n_2 and c_2 such that $g(n) \leq c_2 h(n)$ for all $n \geq n_2$.

Let n_0 be the maximum of n_1 and n_2 and let $c = c_1 c_2$. Then, for all $n \geq n_0$ $f(n) \leq c_1 g(n)$ and $g(n) \leq c_2 h(n)$, so $f(n) \leq c_1 c_2 h(n) = ch(n)$. Therefore, $f(n) = O(h(n))$.

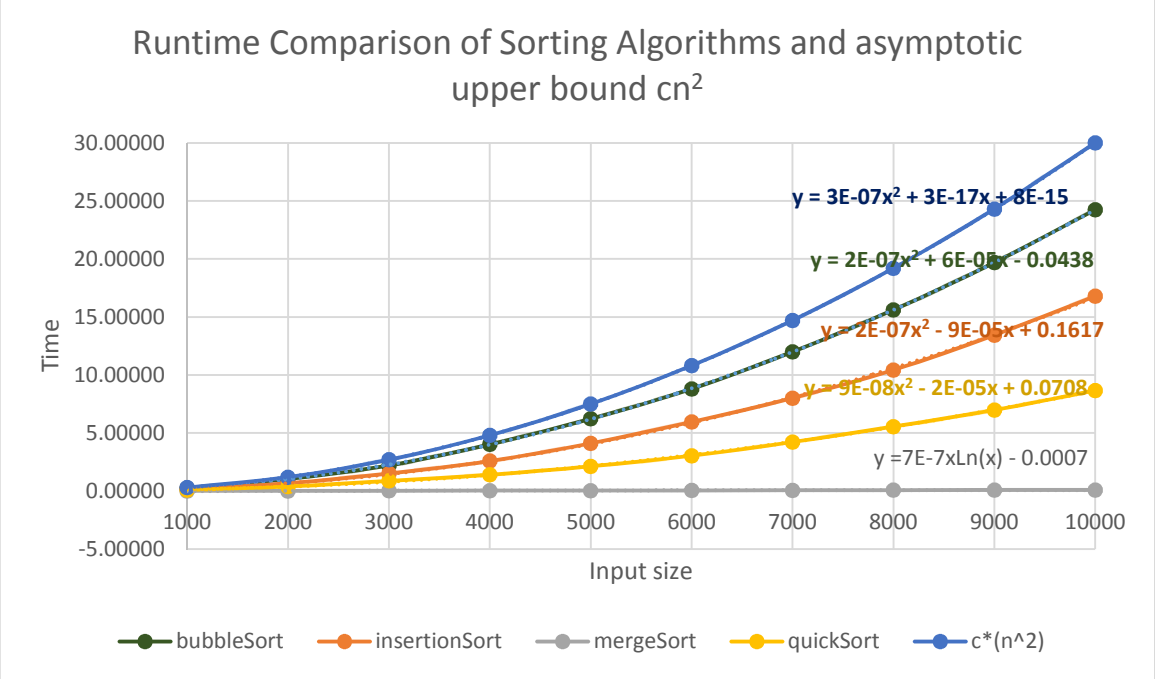
Note:
For the solution of this problem, you need to use the definition of Big-Oh as we did above. You cannot prove this problem by using the limits rule. The limits rule provides sufficient conditions, not necessary conditions. That is,
IF you know that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and that this limit is 0, THEN you can conclude that $f(n) = O(g(n))$.
But it doesn't necessarily work the other way around. That is, if you know that $f(n) = O(g(n))$ you cannot conclude that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ even exists.

Problem 5: (Solutions by Yun Lu)

Time(sec) / n	*bubbleSort*	*insertionSort*	*mergeSort*	*quickSort*	$c_1 n^2$	$c_2 n \log(n)$	***Random_quickSort***
1000	0.26260	0.18678	0.00734	0.08859	**0.3**	**0.007973**	**0.00461**
2000	1.04449	0.67136	0.01513	0.38994	**1.2**	**0.017545**	**0.01322**
3000	2.20468	1.51443	0.02451	0.87549	**2.7**	**0.027722**	**0.01863**
4000	4.02151	2.58445	0.03525	1.40696	**4.8**	**0.038291**	**0.02891**
5000	6.22779	4.11279	0.04501	2.13173	**7.5**	**0.049151**	**0.03450**
6000	8.80387	5.95739	0.05439	3.04104	**10.8**	**0.060244**	**0.03957**
7000	11.99891	8.00732	0.06489	4.22379	**14.7**	**0.07153**	**0.05278**
8000	15.60927	10.43232	0.07623	5.55496	**19.2**	**0.083981**	**0.06729**
9000	19.68651	13.43139	0.08703	6.98035	**24.3**	**0.094577**	**0.07174**
10000	24.24205	16.79881	0.09781	8.66589	**30**	**0.106302**	**0.07912**

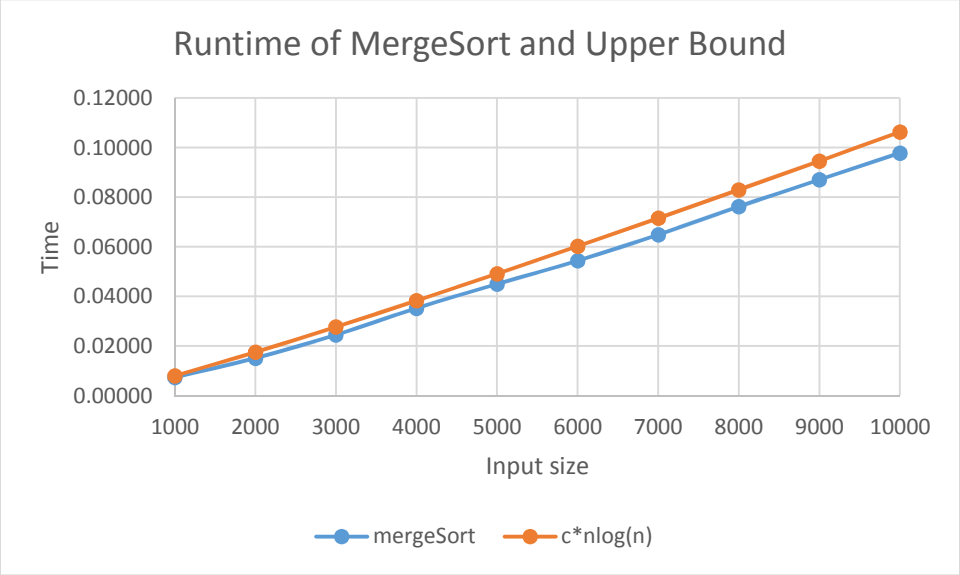
Table 5-1: Input sizes and corresponding runtimes for different sorting algorithms. Here c_1 = 0.0000003 and c_2 = 0.0000008; c1 n2 and c2 n log(n) are showing here for illustration purposes only. Throughout these homework solutions, log base e is used (that is, ln). Random quickSort was not part of the HW, but it is included here so that you can learn this variant of quicksort (where the pivot element is chosen at random).

Plot 5-1 illustrates the fact that bubbleSort, insertionSort, mergeSort, and quickSort are $O(n^2)$.



Plot 5-1: Runtime (in seconds) VS input size for different sorting algorithms. Here c = 0.0000003.

Plot 5-1a shows a tighter upper bound for mergeSort, c*n*log(n).



Plot 5-1a: Runtime (in seconds) of MergeSort and asymptotic upper bound c*n*log(n), c = 0.0000008.

The plots above provide trend lines calculated with Excel and with Matlab, for illustration purposes only. Table 5-2 shows the asymptotic growth behavior of the sorting methods under consideration, as discussed in the textbook (Chapters 2 and 7), and on the website that provides the Python implementation of the sorting methods used in this homework:
http://interactivepython.org/courselib/static/pythonds/SortSearch/sorting.html

	Asymptotic Upper Bounds
BubbleSort	$O(n^2)$
InsertionSort	$O(n^2)$
MergeSort	O(n log(n))
QuickSort	$O(n^2)$

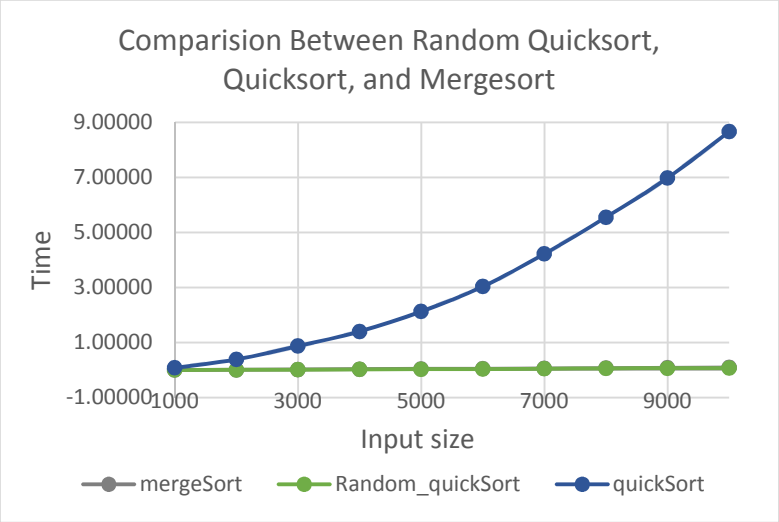
Table 5-2: Asymptotic upper bounds of different sorting algorithms

Notes:

- In the homework, the input list is in decreasing order. After sorting, the list will be in increasing order. This provides a worst case scenario for (most of) these search methods.
- Note that in this HW solutions, we added a variation of quicksort, called Random quicksort here, by selecting one of the elements from list[first] and list[last] at random. We then, exchange this elements with list[first] so that the original code will use this randomly selected element as the pivot element.

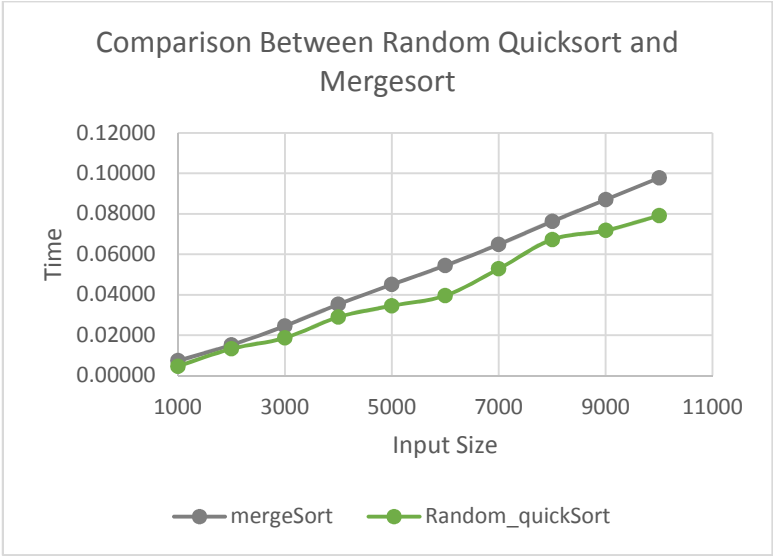
```
position = random.randint(first, last)
temp = alist[first]
alist[first] = alist[position]
alist[position]= temp
```

The running time for random quicksort, quicksort and mergesort are shown below:



Plot 5-2: Runtime comparison for random quickSort, quickSort and mergeSort. The curves for random quicksort and mergeSort overlap in this plot. See Plot 5-3.

Since the curves for random quicksort and mergeSort overlap in Plot 5-2, we depict these curves a more refined level of granularity in Plot 5-3. It is worth noticing that since random quicksort is a randomized algorithm, its running time will vary based on the pivot selected in each recursion. But, we still could see that the running time is O(nlogn).



Plot 5-3: Runtime comparison of random quicksort and mergesort

The table and plot below contain runtime values obtained running the same code on a CS Unix server, a more powerful machine than my own laptop.

	bubbleSort	insertionSort	mergeSort	quickSort
1000	0.16	0.11	0	0.06
2000	0.6	0.43	0.01	0.22
3000	1.47	0.99	0.02	0.48
4000	2.61	1.77	0.02	0.86
5000	4.07	2.79	0.03	1.33
6000	5.86	4.02	0.03	1.92
7000	7.97	5.48	0.04	2.6
8000	10.43	7.16	0.05	3.41
9000	13.17	9.08	0.05	4.31
10000	16.28	11.22	0.06	5.32