# Quiz 2 – CPSC 331 – Fall 2014 – Asymptotic Analysis

1. When using asymptotic analysis to compare the runtime of various algorithms we count the number of operations each algorithm performs. Which of the following would **NOT** be considered a single operation?
	1. A comparison of integers, such as x < 10.
	2. Assigning a value to a variable, such as y = 10.
	3. Inserting an element into a list, such as list1.insert(elem1).
	4. Returning an object from a method, such as return list.
2. The asymptotic runtime of an algorithm is usually defined as a function over
	1. The number of operations in the algorithm.
	2. The number of lines of code in the algorithm.
	3. The size of the input for the algorithm.
	4. Any of the above can be used.
3. For some algorithm it is known that, on input size , the runtime of can be expressed as for some positive constants and . This means that
	1. We do not have enough information to determine asymptotic runtime.
4. Algorithm runs in and algorithm in . Both are worst case asymptotic run times. Which of these two is most efficient?
	1. Algorithm is more efficient than algorithm especially when input size is large.
	2. Algorithm is more efficient than algorithm especially when input size is large.
	3. Algorithm will always run faster than algorithm on any input size and is considered more efficient.
	4. Algorithm will always run faster than algorithm on any input size and is considered more efficient.
5. We must sort more than a million items as quickly as possible. We have narrowed our choice of algorithms to either MergeSort or Quicksort. Both algorithms run in . We have implemented both algorithms and created a sample data set. To ensure our empirical analysis accurately shows which algorithm is best for our algorithm which of the following must we do first?
	1. Optimize the code in both implementations.
	2. Verify that our data set reflects a realistic scenario without unnecessary bias for one algorithm.
	3. Ensure that both algorithms are run on the same computer with a similar workload.
	4. All of the above must be done.
6. In the **best** case, how many operations will we perform to find the item in a list using **linear search**?
	1. The item will be the first element in the list, resulting in a constant number of operations.
	2. The item will be in the middle of the list and we’ll do operations for some positive constant .
	3. The item will be at the end of the list and we’ll do operations for some positive constant.
	4. It is not possible to determine how many operations we’ll perform in the best case.
7. In the in the **worst** case, how many operations will we perform to find the item in a list using **binary search**?
	1. The item will be the last one we consider. Since we cut the list to consider in half after each item considered this will take at worst operations for some positive constant .
	2. The item will be the first one we consider and we’ll do a constant number of operations.
	3. The item will be at the end of the list and we’ll do operations for some positive constant .
	4. It is not possible to determine how many operations we’ll perform in this case.
8. Consider the code for function sum below and let and be positive constants. Which is the best expression of the worst case runtime of sum?
public int sum(int n) {
 int sum = 0;
 for (int i = 1; i <= n; i++) {
 sum = sum + i;
 }
 return sum;
}
	1.
9. Which best expresses the worst case asymptotic runtime of the function sum from the previous question?
10. Consider again the algorithm for MergeSort.

Assume that arraycopy and merge each run in Θ(n) in the worst case. Which of the following recurrence relations best describes the worst case runtime of MergeSort?

 public Element[] mergeSort(Element[] list) {

 if (list.length <= 1) return list;

 int mid = list.length/2;

 Element[] list1 = new Element[mid];

 System.arraycopy(list, 0, list1, 0, mid);

 list1 = mergeSort(list1);

 Element[] list2 = new Element[list.length - mid];

 System.arraycopy(list, mid, list2, 0, list2.length);

 list2 = mergeSort(list2);

 return (merge(list1, list2));

 }

* 1. for .
	2. for .
	3. for and some positive constant .
	4. for and some positive constant .