Question - 1

Equals

SCORE: **5 points**

Which of the following statements are true, regarding equality in Java?

- equals and == are synonyms (i.e. they are exactly the same)
- equals can be overridden by any class
- equals must be consistent with hashCode and, if a class implements Comparable, then it must be consistent with compareTo.
- the result of calling equals does not have to depend on the value of its parameter.
- x.equals(y) is false when y is null but true when x is null.

Question - 2

Complexity analysis (10 marks)

SCORE: **5 points**

Analyze the experiment that you have performed on an algorithm for which you don't have the source code. All you can do is to change *N*, the number of elements to be processed, and time the result. Here are the timings that you observed:

N	T
1000	3.4
2000	9.8
4000	23.8
8000	65.8
16000	222.6
32000	601.1

Determine the complexity of the algorithm.

Note that this question counts for 10 marks, not the 5 that HackerRank awards.

- O(1)
- O(log N)
- O(N)
- O(N log N)

- [x] O(N^1.5)
- [ ] O(N^2)
- [ ] O(N^3)

Question - 3

SCORE: 16 points

Return on investment

You are a software manager responsible for a product runs a kind of lottery. It relies on an algorithm to find the top 10% of scores from the entries filled out be each participant. The developer who coded the critical part of this product, who (by the way) was fired several months ago for incompetence, used *quicksort* followed by *take* to achieve the result. This worked well during alpha testing, beta testing, and even in early production runs where N was typically 100,000. But the lottery has proved very popular and approximately one million people are entering it. Your company is now having to pay a penalty for poor performance (the product runs on AWS and every extra second costs money).

A developer has estimated that they can rewrite the method using quick-select, testing it thoroughly, within two weeks. Including overhead, you estimate this to cost the company $8,000. On the other hand, each day the run (which is compute-bound) is costing $1000. The lifetime of the product is expected to be a matter of weeks at most before it is replaced.

Calculate the number of days (counting from when the new version is deployed) before the company breaks even, assuming you choose to implement quickselect. You may assume that the number of operations for quicksort is $2\,n\,ln(n)$ on average. And for quickselect, the number of operations is $3.4\,n$ on average.

You might observe that you really don't need to know anything about algorithms or data structures to answer this question because I have given you all the formulas that you need. But it's important to be able to perform these "back of the envelope" calculations reasonably accurately.

Question - 4

SCORE: 5 points

System sorts (10 marks)

By its very nature, a system sort knows very little about what it's sorting so it tends to use a "one-size-fits-all" strategy. However, there is one piece of information that it does know: whether it is sorting primitives or objects.

In Java, the system sort for primitives is (dual-pivot) *quicksort*. For objects, it's *timsort* (a variation on merge-sort which tends to linear performance when the array to be sorted is already in order).

Select the factors below that you believe **contribute to this decision** (don't select factors that are *either* untrue *or* are true but don't affect the decision used by the Java library):

- [x] Objects (as opposed to primitives) are often in partial order because lists of objects are typically made up of a previously sorted list together with some (new) additional unsorted items.

Comparison of objects tends to be much slower than comparison of primitives, while exchanging (swapping) them takes the same amount of time.

Primitives have only one natural key, but objects can have many keys (up to N!) where N is the number of independent fields. Thus stability is much more important for objects because a sort is more likely to be based on an alternative key.

In the worst case, quick sort can be O(n^2) but merge sort has worst case of O (n lg n)

Arrays of primitives are more cache-friendly than arrays of object references (the object references themselves are cache-friendly but the objects, which must be consulted for purposes of comparison) will typically not be in-cache.

Mergesort can use insertion sort (instead of recursion/merge) for small values of N

Question - 5

SCORE: 5 points

Quick sort (20 marks)

Please identify the true statements regarding quicksort from the list below:

Quick sort has best and average complexity of O(N log N) but its worst case complexity is O(N^2).

Quick sort is the system sort in Java.

Quick sort is stable

Quick sort is in-place

Quick sort is sensitive to the choice of pivot and, for this reason, we typically precede quick sort by a random shuffle.

Quick sort was developed as a derivative of merge sort, designed to avoid the extra memory of merge sort.

Quick sort does use some extra memory: proportional to log N (rather than N).

The total number of operations in the inner loop is slightly more than for merge sort.

Question - 6

SCORE: 5 points

Simple data structures (8 marks)

Which of the following are true?

The best and simplest implementation for a stack is a linked list.

The best and simplest implementation for a bag is a set.

- ○ A queue requires two linked lists for its implementation.
- ◉ A priority queue can be implemented with an (unordered) array.

Question - 7

SCORE: 21 points

Reduction

Problem Statement

Fill in the blanks. The answers are mostly simple English words. There are no tricks here. Just plain common sense.

Reduction is a technique in developing algorithms whereby a problem is transformed into one or <blank 1> sub-problems for which we already have a <blank 2>. Once those problem(s) are solved the result is transformed back into the domain of the original <blank 3>. As long as the total <blank 4> to perform the two transformations, and each of the sub-problems, is less than that required for the original problem, we will come out ahead. However, reduction can also be used to <blank 5> our code, even though we know that it may end up taking longer than the best solution to the original problem. For example, suppose we need the five largest elements from a list of 100 (perhaps to give prizes after an exam). One efficient way to do this is to use <blank 6> which runs in <blank 7> time. But, since 100 is not a large number, we could simply sort the list and then take the first five elements.

Answers

<blank 1> : [more]
<blank 2> : [solution]
<blank 3> : [problem]
<blank 4> : [work, effort, time]
<blank 5> : [simplify, facilitate, make easy]
<blank 6> : [quickselect, quick select]
<blank 7> : [linear, O(N), O(n), N, n]

Question - 8

SCORE: 84 points

PriorityQueue

Be sure to read this entire problem description **carefully**. I promise you that you will easily recoup the time spent reading and understanding. I strongly advise you to leave this question until you are finished or almost finished with the other questions. The probability that you may get bogged down in coding is too great. There are too many points available in the other questions to skip them.

Complete the five stubs to implement a priority queue using a binary heap. The priority queue is defined at construction time to be either a maximum or a minimum PQ. There are no arbitrary limits on the size of this PQ, but its capacity is fixed at construction time. If an element is "given" to a PQ which is at capacity, the "least" element will first be removed so that there is room for the new element. Any object can be placed in the PQ because we pass in a comparator to enable comparison between any two objects.

The method names for this PQ are a little different from what's in the book. We use *give* and *take* instead of *insert* and *delMax*. We use *sink* and *swimUp* for the internal heap restoration methods. And we use *unOrdered* instead of *less* (this is because our PQ can serve as either a maxPQ or a minPQ).

We also use *parent(k)* and *firstChild(k)* instead of *k/2* and *2*k*. This is because I don't like logic that uses arbitrary "magic"

formulas. Although I didn't generalize the PQ to be an *n*-ary heap, it could easily be done using these methods. If the compiler isn't smart enough to optimize these calls (it should be) then, if you were in production, you would be entitled to replace those calls with the more efficient integer operations. For now, just use *parent(k)* and *firstChild(k)*. I will deduct 1 point (total) if you (prematurely) optimize these invocations.

Other than the extra generality of this PQ implementation, there's nothing tricky in the code. It's just like I showed you last week. One thing to keep in mind is that, in order to simplify the *parent* and *firstChild* calculations, we reserve the 0th element of the *binHeap* array. That's to say, the binary heap actually starts at element 1 of its array (not at element 0). Again, that is just as it is in the book.

There are many comments in the code which should help you out. But you should be aware that we reserve the first element of the *binHeap* array just to make the parent/child arithmetic faster.

The unit tests are your friend and, if you use them wisely, you should be able to complete the programming in less than 20 minutes. It is *strongly* advised to complete the methods and test them in the following sequence:

1. unordered
2. swim/sink
3. give/take

Question - 9

SCORE: 26 points

Binary search (hard question)

---

1. What conditions are required for binary search to operate?
2. Data structure:
    1. What is the most efficient underlying data structure for binary search?
    2. Why?
3. Explain, with reference to *geometry*, how it is that binary search is able to reduce search time from *O(N)* to *O(log N)*.