C(n,2) =n!/(2!(n-2)!) = n(n-1)/2 = 1+2+3+...+(n-1). The combination formula and sum formula coinciding is not purely coincidental.

Elementary Sorting Algorithms

1. Which method runs faster for an array with all keys identical, selection sort or insertion sort? Why?

1) Insertion. Selection makes n!/(2(n-2)!)=n(n-1)/2 comparisons no matter the keys, while insertion makes at most the same number of comparisons and in this case makes the minimum which is n-1. Neither make any swaps of course. You may want to rewrite the first 2 questions. I thought you meant identical between two the methods. If you receive answers like my second answers to these first 2 questions, that is probably why.

1) Insertion is at most O((n^2)/2)=O(n^2) while selection is equal to the same since selection's inner loop iterates over all of the numbers up to the outer loop's index while insertion only does so at maximum (depending on initial order). This insertion is generally faster.

2. Which method runs faster for an array in reverse order, selection sort or insertion sort? Why?

2) The difference here from the first question is that comparisons for insertion is now maximum, meaning equal to the aforementioned invariant number of comparisons that selection makes; and that swaps are maximum for both but n for selection and the same as number of comparisons for insertion. Though the time complexity is the same there are more swaps for insertion.

2) Reverse order makes no difference except in the case of initial order (or reverse of) being the one where insertion's iterations are equal to selection's (which is when the order is complete opposite).

3. Suppose that we use insertion sort on a randomly ordered array where elements have only one of three values. Is the running time linear, quadratic, or something in between?

Pigeonhole pinciple tells us that a combo from a finite set of keys will necessarily have repetitions after the combo reaches the size equal to the size of set of keys. , the ratio of repetitions necessarily increases with larger array, while normally the number of consecutive repetitions increases with larger array while ratio of consecutive repetitions stays the same. The former means
Another difference is how many keys on average that need to be swapped is lower compared to when there are more than three keys with selection being more significantly lower. For insertion there are on average less comparisons.
Despite these differences, the runtime for insertion is still quadratic. For example, 3213213... is (1/3)*2/3+(1/3)*1/3=1/3 times the max time complexity (reverse order, no repetitions) meaning on the order of (n^2)/6 which is still O(n^2). And, this isn't as bad 33...22...11...

4. A colleague suggests you use insertion sort to sort a singly linked list in ascending order. Why is this a bad idea? How does insertion sort's runtime complexity change if you were to use it to sort a linked list?

There are the same number of comparisons, but you don't need to swap for every comparison except the final (insertion) swap. You can create a new node with the data of the data of the head node of the unordered list and insert it into the correct spot (after a sequence of comparisons) of the ordered list, and then delete said head. The time complexity is still O(n^2) and the space complexity is still O(1). If we insist that swapping for every comparison before the insertion is necessarily part of what defines insertion sort, then time complexity is O(n^3).

5. A colleague suggests you use selection sort for h-sorting in shellsort? Why is this a bad idea?

After each h-sort (after each decrease of h), the sorts are better partially ordered, and insertion is faster for partially ordered.

6. A clerk at a shipping company is charged with the task of rearranging a number of large crates in order of the time they are to be shipped out. Thus, the cost of compares is very low (just look at the labels) relative to the cost of exchanges (moving the crates). The warehouse is nearly full: there is extra space sufficient to hold any one of the crates, but not two. What sorting method should the clerk use? Justify your answer.

Selection sort: it minimizes the exchanges down to at most the number of crates.

Merge Sort

7. A colleague thinks that input arrays to the merge operation of merge sort don't need to be in sorted order. Is your colleague right or wrong? Why?

Wrong. During each merge, whether 2, 3, n-way, the first element of each input arrays are compared.

If the input arrays are (6, 1, 8, 0) and (5, 3, 7, 3), then you get (5, 3, 6, 1, 7, 3, 8, 0), which is far from ordered.

8. Give traces showing how the keys `69 65 83 89 81 85 69 83 84 73 79 78` are sorted with top-down mergesort.

69 65 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 69 81 83 85 84 73 79 78
65 69 69 81 83 83 85 89 84 73 79 78
65 69 69 81 83 83 85 89 73 84 79 78
65 69 69 81 83 83 85 89 73 84 78 79
65 69 69 81 83 83 85 89 73 78 79 84
65 69 69 73 78 79 81 83 83 84 85 89

9. Give traces showing how the keys `69 65 83 89 81 85 69 83 84 73 79 78` are sorted with bottom-up mergesort.

69 65 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 73 84 79 78
65 69 83 89 81 85 69 83 73 84 78 79
65 69 83 89 81 85 69 83 73 84 78 79
65 69 83 89 69 81 83 85 73 84 78 79
65 69 83 89 69 81 83 85 73 78 79 84
65 69 69 81 83 83 85 89 73 78 79 84
65 69 69 73 78 79 81 83 83 84 85 89

10. An array contains n numbers, and you want to determine whether two of the numbers sum to a given number k. For instance, if the input is 8, 4, 1, 6 and k is 10, the answer is yes (4 and 6). A number may be used twice.

(a) Describe an $O(n^2)$ algorithm to solve this problem.

Outer loop iterates i over the array of n numbers. Inner loop iterates j over the array.
Whether the inner loop excludes prior i's or not doesn't affect O(n^2), though excluding would
obviously be faster. In the inner loop, if i + j == k then break, and the same after the inner loop
but inside the outer loop. Outside the outer loop, if i + j == k then print "yes", else print "no".

(b) Describe an $O(n \log_2 n)$ algorithm to solve this problem. Hint: sort the items first. After doing so,

you can solve the problem in linear time.

Merge or tim sort. Then let the outer loop iterate i from 0 to n-1 while a[i] + a[j] < k. The inner loop iterates j from m to 0 while a[i] + a[j] > k. m = j. After the outer loop, if i + j = k then print "yes", else print "no". Despite the explicit bounds of the loops, they are implicitly bounded such that the sum of iterations of the two loops is at most n.

11. Suppose instead of dividing in half at each step of merge sort, you divide into thirds, sort each third, and combine using a 3-way merge. We call this new sorting method 3-way merge sort. What is the time complexity of 3-way merge sort? Is it worth it to use 3-way merge sort over 2-way merge sort?

Similarly to how time complexity of 2-way is O(n log_2 n), the time complexity of 3-way is thus O(n log_3 n). And, time complexity is invariant with respect to log base. There will be fewer swaps and more comparisons, but with greater difference (within one order of magnitude) in number of swaps. Whether it is worth it depends on the differenct in costs.

Quick Sort

12. Show the result of standard quicksort (with no optimizations or improvements) partitioning a subarray containing `69 65 83 89 81 85 69 83 84 73 79 78`.

65 69 83 89 81 85 69 83 84 73 79 78

13. Show a trace of how standard quicksort (with no optimizations or improvements) sorts an array containing `69 65 83 89 81 85 69 83 84 73 79 78`. For the purposes of this exercise, ignore the initial shuffle.

69 65 83 89 81 85 69 83 84 73 79 78
65 69 83 89 81 85 69 83 84 73 79 78
65 69 83 78 81 85 69 83 84 73 79 89
65 69 83 78 81 79 69 83 84 73 85 89
65 69 83 78 81 79 69 83 73 84 85 89
65 69 73 78 81 79 69 83 83 84 85 89
65 69 73 78 81 79 69 83 83 84 85 89
65 69 73 69 81 79 78 83 83 84 85 89
65 69 69 73 81 79 78 83 83 84 85 89
65 69 69 73 79 78 81 83 83 84 85 89

14. Explain what happens when standard quicksort (with no optimizations or improvements) is run on an

The only swaps are pivot swaps (since strict inequalities are used), and it only meaningfully happens with the array having items with only two distinct keys. greater of the two. The j index stops on a lesser for a greater pivot, and doesn't stop until the last key for a lesser pivot. When the lesser of the two is the pivot and the following key is another lesser, the scanner from the left ends on the last consecutive lesser from the pivot and the scanner from the right runs all the way through, which results in a swap of two lessers. When a lesser is a pivot and is followed by a greater, the scanner from left stops immediately and the scanner from right ends up on the same key, meaning no swaps occur. All of this ends in the lessers all being on the left and greaters being on the right, with O(n^2).

15. On average, standard quicksort (with no optimizations or improvements) runs in $O(n \log_2 n)$ time. However, its worst case time complexity is still listed as $O(n^2)$. Under what scenarios will quick sort perform so poorly?

    The above scenario is one. Another is when all the keys are equal. Another is when they are already sorted.

16. What benefits does 3-way quicksort provide over the traditional 2-way quicksort? Remember that 3-way quicksort is a flavor of quicksort where partitioning is done based on three relational buckets (less than, equal to, and greater than the subarray).

    It is a better choice when there are a lot of repetitions.