Assignment 10 - Sorting & Searching Ordered Data

Q1.

i)

For a sorted file with size n, assume elements prior the index i is sorted, and element at i is about to be “inserted” to the sublist, since it’s already sorted, only one comparison is needed for that element. Therefore for n elements, **the number of the comparison is n.**
No interchange will take place, because the file is already sorted.

ii)

If the file is sorted in reverse order, given the assumption for i), inserting i-th element requires comparisons of the element with the entire sorted sublist, so in number that will be i comparisons for i = [1, n-1]. Hence for sorting a reversely sorted file, requires **1 + 2 + … + n-1 = n(n-1)/2 comparisons.**
Additionally, every element checked needs to move to the first position and all other elements from the sublist need to be shifted backward. So the number of interchanges for i-th element will be i for i = [1, n-1]. **In total, it would be 1 + 2 + … + n-1 = n(n-1)/2 interchanges**

iii)

For a file in which x[0], x[2], x[4] … are the smallest elements in sorted order, and x[1], x[3], x[5]... are the largest elements in sorted order:

For sorting the given example [3 14 5 15 9 18 11 19], numbers of comparisons and interchanges as below:

Index i of the element sorted	# of comparisons	# of interchanges	Interim result
1	1	0	[3 14 | 5 15 9 18 11 19]
2	2	1	[3 5 14 | 15 9 18 11 19]
3	1	0	[3 5 14 15 | 9 18 11 19]
4	3	2	[3 5 9 14 15 | 18 11 19]
5	1	0	[3 5 9 14 15 18 | 11 19]
6	4	3	[3 5 9 11 14 15 18 | 19]
7	1	0	[3 5 9 11 14 15 18 19 |]
i	1 for odd i, i/2 + 1 for even i	0 for odd i. i/2 for even	

So the total number of comparisons is 1 + 2 + 1 + 3 + 4 + 1 … = n/2 + [2 + 3 + … + (n-1)/2 +1]

$$= \sum_{i=1}^{n-1} f(i) \quad , \quad \text{for} f(i) \begin{cases} 1, \text{ if i is odd} \\ \frac{i}{2} + 1 \text{ , if i is even} \end{cases}$$

The total number of interchanges is:

$$0 + 1 + 0 + 2 + \ldots = \sum_{i=1}^{n-1} g(i) \text{ , for } g(i) = \begin{cases} 0, \text{ if i is odd} \\ \frac{\text{i}}{2}, \text{ if i is even} \end{cases}$$

Q2:

i) For a sorted file [1 2 3 4 5 6 7 8 … n]

If the gap is 2, the file is splitted to [1 3 5 …] and [2 4 6 …]. Sorting subgroups, which are sorted as well, requires n/2 comparisons and 0 interchanges for each group.

After changing the gap to 1, sorting requires extra n comparisons. **Therefore, in total 2n comparisons and 0 interchange are needed.**

ii) For a reversed order list [8 7 6 5 4 3 2 1]

While the gap is 2, the file is splitted as [8 6 4 2] and [7 5 3 1], which are reversely sorted. From the result of Q1 ii), insertion sorts of these two groups cost m(m-1)/2 * 2 comparisons and m(m-1)/2 * 2 interchanges for m = n/2. And the result will be [2 1 4 3 6 5 8 7].

Then for the gap = 1, insertion sort costs 1 comparison and 1 interchange for each insertion of even indexed elements, so n/2 in total.

Summing up costs of two level shell sorts, the total numbers of comparisons and interchanges will be:

$$\frac{m(m-1)}{2} \cdot 2 + \frac{n}{2} = m(m-1) + m = m^2 = \frac{n^2}{4}$$

iii) For a mixed ordered file [3 14 5 15 9 18 11 19]

While the gap is set as 2, the sub lists will be [3 5 9 11] and [14 15 18 19], which are sorted. So insertion sorts at this level cost n/2 * 2 = n comparisons and 0 interchanges. And the base list is not changed.

Then the gap is set to 1, since we know the from the result of Q1 iii), the total cost of the shell sort will be:

Comparison: $$n + \sum_{i=1}^{n-1} f(i) \quad \text{for } f(i) = \begin{cases} 1 & i \text{ is odd} \\ \frac{i}{2} + 1 & i \text{ is even} \end{cases}$$

Interchange: $$\sum_{i=1}^{n-1} g(i), \text{ for } g(i) = \begin{cases} 0, \text{ if i is odd} \\ \frac{\text{i}}{2}, \text{ if i is even} \end{cases}$$

Q3

a. m = n and a[i]<b[i] < a[i+1], e.g a=[6 9 12 15 29 37] and b = [8 10 14 25 33 45]

There will be 2 comparisons for each element in a and b except the first element of a and last element of b. eg: 6 vs 8, 9 vs 8, 9 vs 10, … 29 vs 33, 37 vs 33, 37 vs 45. So the total number of comparisons will be **(2m -1 + 2n-1)/2 = (4n -2)2 = 2n -1**

b. m = n and a[n] < b[1], e.g. a = [2 5 9] and b= [12 14 16]

Since the all elements from a is smaller than the first element of b and a & b are sorted, merging two files is actually merging all elements of a first then merging b. So comparisons will be: 2 vs 12, 5 vs 12, 9 vs 12. **In general the total number will be n.**

Q4.

a. m=n and a[n/2] < b[1] < b[m] < a[(n/2)+1]
Eg. a = [2 5 7 55 61 72] , b = [9 15 17 21 29 46]

In this situation, a can be splitted as a1 = [2 5 7] and a2 = [55 61 72], then the merging will be converted as merging a1 and b first, then a2 and b.

For merging the first part of a, since a1[i] for any i are smaller than b[1], it takes n/2 comparisons to merge. Then since b[i] for any i are smaller than a2[1], so merging costs m comparisons. **In total, n/2 + m = 1.5n comparisons are required.**

b. m =1 and b[1]< a[1]
E.g a = [2 3 4] b = [1]

After comparing b[1] and a[1], b[1] is merged to the new file, and no more comparisons are required. **So 1**

c. m = 1 and a[n]< b[1]
E.g a = [1 2 3] b = [100]

Since the only element from b is larger than any element from a, so comparisons between a[i] for any i with b[1] is required. **Therefore in total n comparisons are performed.**

Q5.
If elements of files are randomly distributed, finding the largest or smallest of a set n requires n-1 comparisons because the program can linearly scan the list and store the largest or smallest element that has been read so far. While accessing each element, we can compare the largest so fat with that element. Therefore n-1 comparisons are needed in this case.

```java
public static int findLargest(int[] arr){
    int res = arr[0];

    for (int i = 1; i< arr.length; i++){
        if (arr[i]>res) res = arr[i];
    }

    return res;
}

public static int findSmallest(int[] arr){
    int res = arr[0];

    for (int i = 1; i< arr.length; i++){
        if (arr[i]<res) res = arr[i];
    }

    return res;
}
```

If the file has sorted in specific order, no comparison is needed.

Q6.

a. the sequential search needs to scan the entire table. For an unordered table, at the specific point the element scanned exceeds the key target then no matching records can be concluded. So it's more efficient to operate sequential searches on ordered tables than on random ordered tables

b. In this case, the efficiency of the sequential searching is not affected by whether the table is ordered or not.

Q7.

a. Same as Q6 b), the efficiencies of using sequential search are the same for both cases.
b. For an ordered table, once the first appearance of the key target is sought, consecutive multiple elements are the key targets too. While for an unordered table, sequential search still requires more searching on following records since they are not in order. Therefore, for ordered tables the use of sequential search is more efficient.