**Assignment 10 – Sorting - Searching Ordered Data**

*Write pseudo-code not Java for problems requiring code. You are responsible for the appropriate level of detail. For all the questions in this set, assume you are working in arrays.*

1. **How many comparisons and interchanges (in terms of file size n) are performed by Simple insertion sort for the following files:**

**i) A sorted file  
ii) A file that is sorted in reverse order (that is, from largest to smallest)  
iii) A file in which x[0], x[2], x[4]... are the smallest elements in sorted order, and in which x[1], x[3], x[5]... are the largest elements in sorted order, e.g. [ 3 14 5 15 9 18 11 19 ].**

If we are going by the way that insertion sort is implemented in the lecture, then we are going to insert from the right, and the temp list will be ordered from smallest to largest. This means that if the new value is larger than any of the values already found in the sorted array, there will only be one comparison.

1. For this problem, take an example [1,2,3,4,5]. We will take 1 and put it in the first section. Then we move to 2, compare it to 1, and do nothing. Then we move on to 3, compare to 2, and do nothing. This occurs no matter the size of the array, since every value is larger than the one preceding it. That means there will be n-1 comparisons and 0 interchanges.
2. For this problem, take an example [5,4,3,2,1]. We will take 5 and put it in the first spot. Then we will take 4, compare it to 5, and interchange 4 and 5. Then we take 3, compare it to 5, interchange 3 and 5, compare 3 and 4, interchange 3 and 4. Here is a table of the step and the number of comparisons and interchanges.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Number | Comparisons | Interchanges |
| 1 | 5 | 0 | 0 |
| 2 | 4 | 1 | 1 |
| 3 | 3 | 2 | 2 |
| 4 | 2 | 3 | 3 |
| 5 | 1 | 4 | 4 |

So we have a series 0+1+2+…+n-1, which is (n-1)(n-2)/2. The total cost is then (n-1)(n-2), since we have to add the cost of comparisons and interchanges.

1. This one is a little tricky, so let us build a table to see.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Number | Comparisons | Interchanges |
| 1 | 3 | 0 | 0 |
| 2 | 14 | 1 | 0 |
| 3 | 5 | 2 | 1 |
| 4 | 15 | 1 | 0 |
| 5 | 9 | 3 | 2 |
| 6 | 18 | 1 | 0 |
| 7 | 11 | 4 | 3 |
| 8 | 19 | 1 | 0 |

It seems that for this example, half of the dataset has only 1 comparison and 0 interchanges. This makes sense, as we saw earlier that when the dataset is sorted, there is only 1 comparison and 0 exchanges for every number. Half of the dataset follows that rule. The other half seems to follow the increase in which everything is reverse sorted, except that there is one less interchange than the comparison. So I think the formula would be ½\*n(for the half of the dataset that is larger) + ((1/2\*n-1)(1/2\*n-2)/2) for comparisons. For interchanges, it would be (1/2\*n-2)(1/2\*n-3), since there is one less interchange than comparisons, and the other half is 0. The total would be

1. **How many comparisons and interchanges (in terms of file size n) are performed by Shell Sort using increments 2 and 1 for the following files:**

**i) A sorted file  
ii) A file that is sorted in reverse order (that is, from largest to smallest)  
iii) A file in which x[0], x[2], x[4]... are the smallest elements in sorted order, and in which x[1], x[3], x[5]... are the largest elements in sorted order, e.g. [ 3 14 5 15 9 18 11 19 ].**

**i)** If we take an example [1,2,3,4,5,6], then the two subfiles using k=2 will be [1,3,5] and [2,4,6]. Then an insertion sort will be performed on each subfile. Since these are in order, we can take the result from our previous work that the comparisons will be n-2 [i.e. (n/2-1+(n/2-1))] and the interchanges are 0. This will return the same array [1,2,3,4,5,6]. Then we go to a subfile of 1, which is [1,2,3,4,5,6] and do insertion sort on it. Since this is the same increasing order, comparisons will be n and interchanges will be 0. This means that, for a sorted file and increments of 2 and 1, the total work done will be n + n -2, or 2n -2 .

**ii)** If we take example [6,5,4,3,2,1], then the two subfiles using k =2 will be [6,4,2] and [5,3,1]. To do an insertion sort, we could take the previous insertion sort example of having the reverse ordered files. There, we found that the amount of work done was (n-1)(n-2). Since we have two partitions of 1/2n, the amount of work done with two subfiles is then 2\*(1/2n-1)(1/2n-2). When we combine our sample file, we will get [2,1,4,3,6,5]. The amount of work done is then

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Number | Comparisons | Interchanges |
| 1 | 2 | 0 | 0 |
| 2 | 1 | 1 | 1 |
| 3 | 4 | 1 | 0 |
| 4 | 3 | 2 | 1 |
| 5 | 6 | 1 | 0 |
| 6 | 5 | 2 | 1 |

We can see that the equation is then 0+1+1/2(n-2)+2\*1/2(n-2). The first two are always 0 and 1, and half of the remaining has 1 comparison. The other half has 2. The number of interchanges are 1/2n. If we add the two together, we get 1+1/2n-1+n-2+1/2n, which is 2n -1. If we add this to the k=2 work, we get 2\*(1/2n-1)(1/2n-2)+2n-1.

**iii)** If we take the example given, our first k=2 subfile will be [3,5,9,11] and [14,15,18,19]. This already sorted, so the comparisons is n-2 and the interchanges will be 0. We then get the same file we inputted, and we do insertion sort on this. This was already covered previously in example 3, and the cost was found to be

We would just have to add a n-2 to this for the additional comparisons done at k=2.

1. **Determine which of the following sorts is most efficient. Consider if the data is small and simple or larger and more complex.**

**a) simple insertion sort   
b) straight selection sort   
c) bubble sort**

For the purposes of this problem, I am going to assume “simple” means mostly in order. For bubble sort, we make a series of pairwise comparisons and switch on the spot. The first pass, the largest number will always be at the end. The second pass, the second largest, and so on. This can take advantage of order in the file, as if no exchanges were made, the method can stop since the file will be in order. The cost is O(n^2).

For straight selection sort, the amount of work is the same no matter what. For simple files, there is no way for the selection sort to stop early. The cost is always O(n^2).

For simple insertion sort, the cost on ordered data is O(n), while the cost for reverse ordered data is O(n^2).

So for simple data (mostly ordered), it appears that simple insertion sort is the best, followed closely by bubble sort. Straight selection sort is the worst, since there is no early out, and it will go through the whole process regardless of whether the data is close to sorted or not.

For complex data, aka more random/reverse ordered, the advantage would have to go to simple insertion sort or bubble sort. Both of these have mechanism built in to stop earlier if a certain criteria is met. For simple insertion sort, it will stop making more comparisons for a number once a comparison fails to make a swap. For a bubble sort, it will stop the method once a step fails to make any swaps. If I had to pick between these two, I would go for the bubble sort, since the average cost is 1/2n^2+1/2n, whereas the insertion sort could be much higher depending on the order of the data.

For larger datafiles, holding the simplicity constant, I would choose insertion sort. Insertion sort has the possibility of reducing the number of comparisons made as well based on the order of the data, whereas bubble sort and quick sort have a pretty linear number of comparisons (n,n-1,n-2,etc.). So for small, simple files, I would choose bubble sort. For large, simple files I would choose insertion sort. For small, complex files I would choose bubble sort. For large, complex files, there really isn’t a great choice. I would be torn between bubble and insertion, but probably insertion sort for the potential comparison savings.

1. **Determine the number of comparisons (as a function of n and m) that are performed in merging two ordered files a and b of sizes n and m, respectively, by the merge method presented in the lecture, on each of the following sets of ordered files:**
   1. **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]**
   2. **m=n and a[n] < b[1], e.g. a =[ 2, 5, 9] and b = [12, 14, 16]**

**a[i] refers the value in position i of file a, etc.**

**a.** The final list will look like [a[1],b[1],a[2],b[2],etc.]. This means that for a m=n of size 6 as in the example, the number of comparisons that will be made is 11, or m+n-1. This is because every number except the last number will be involved in a comparison, or rather that it will lead a comparison. This can be seen through an example of m=n of length 2, where a[1]<b[1]<a[2]<b[2]. There are 3 comparisons that will be made, which is 2+2-1.

**b.** The final list will be [a[1],a[2],a[3],b[1],b[2],b[3]]. This means that all of a[i] will be compared to b[1]. However, after those comparisons are finished, there will be nothing for b[i] to compare to. Thus, the number of comparisons is only n, the length of a.

1. **Determine the number of comparisons (as a function of n and m) that are performed in merging two ordered files a and b of sizes n and m, respectively, by the merge method presented in the lecture, on each of the following sets of ordered files:**
   1. **m=n and a[n/2] < b[1] < b[m] < a[(n/2)+1],**

**e.g. a = [2, 5, 7, 55, 61, 72] and b =[9, 15, 17, 21, 29, 46]**

* 1. **m=1 and b[1] < a[1]**
  2. **m=1 and a[n] < b[1]**

**a[i] refers the value in position i of file a, etc.**

* 1. The finished list will look like [a1,a2,a3,b1,b2,b3,b4,b5,b6,a4,a5,a6]. The number of comparisons will be 3 for the first part of a, and 6 for all the elements in b. Once that is done, however, there will be nothing left for the rest of a to compare to. Thus, the number of comparisons made is 1/2n+m.
  2. This means that there is only one element in b, and that it is less than all of the elements in a. There will be one comparison, b1<a1. Once that is done, there is nothing for A to compare to. Thus, the number of comparisons is one.
  3. Here, we have the only element in b greater than every element in a. This means that every element in A has to be compared with the single element in b. This means that the number of comparisons made is the length of A, which is n.

**For questions 6 – 9, compare the efficiency of using sequential search on an ordered table of size n and an unordered table of the same size for the key *key*:**

1. **If no record with the key *key* is present**

If search is performed on data without the record key, then for both ordered and unordered data, the whole dataset will be searched. This means that the ordered and unordered dataset are equal in efficiency.

1. **If one record with the key *key* is present and only one is sought.**

For an ordered table, there will be efficiency gains if key is small and is towards the beginning of the dataset, and efficiency losses if it is large and towards the end. However, there are ways to mitigate this through indexing and interpolating, but I am not sure if that is being discussed here. For unordered data, it is fairly up to chance if the record will be located in the beginning of the data or at the end. On average, the two should fare similarly in efficiency, since it is not guaranteed where the record will be.

1. **If more than one record with the key *key* is present and it is desired to find only the first**

However, if the data contains multiple records, it is far more likely for the unordered to chance upon a key. The ordered data will not have any efficiency gains since the two duplicates will be next to each other. This means that there is a greater chance that one of the duplicate records will be at the front of the list in an unordered list. The unordered list should gain efficiency with each increase in duplicates.

1. **If more than one record with the key *key* is present and it is desired to find them all.**

Here, we have the opposite. When it is desired to find them all, the unordered dataset search has to loop through the entire dataset every time. Otherwise, there is a chance that it might have missed the value. For ordered, however, it only needs to find the first one, and continue until the record changes. This is because the duplicate values will be next to each other. The sorted dataset will have a huge efficiency gain over the unsorted one.