

Trường Đại Học Quốc Tế - ĐHQG TP.HCM

LAB REPORT

Course: Algorithms & Data Structures LAB 6

Full Name: Trần Minh Phúc

Student's ID: ITCSIU24070

1 PartitionApp.java

1.1 Add counters for the number of comparisons and swaps and display them after partitioning

```
1  public int partitionIt(int left, int right, Long pivot)
2  {
3      int leftPtr = left - 1;           // right of first elem
4      int rightPtr = right + 1;        // left of pivot
5      while(true)
6      {
7          comparisonCount++;
8          while(leftPtr < right &&      // find bigger item
9              theArray[++leftPtr] < pivot)
10             comparisonCount++;
11             ; // (nop)
12         comparisonCount++;
13         while(rightPtr > left &&      // find smaller item
14             theArray[--rightPtr] > pivot)
15             ; // (nop)
16         comparisonCount++; // count final comparison
17         if(leftPtr >= rightPtr){      // if pointers cross,
18             break;                    // partition done
19         } else{                       // not crossed, so
20             swap(leftPtr, rightPtr);  // swap elements
21             swapCount++;
22         }
23     } // end while(true)
24     return leftPtr;                  // return partition
25 } // end partitionIt()
```


1.2 Investigate the relationship between the index of partitioning, the number of comparison, and the number of swaps.

As the code runs, we can infer that the number of comparison is less than the partition index, while the partition index is greater than or equal the number of swaps.

1.3 Compute the average number of comparisons and swaps over 100 runs.

```
1  public static void main(String[] args)
2  {
3      double averageComparisonsIn100Runs = 0;
4      for(int i = 0; i < 100; i++){
5          int maxSize = 16;           // array size
6          ArrayPar arr;               // reference to array
7          arr = new ArrayPar(maxSize); // create the array
8          for(int j=0; j<maxSize; j++) // fill array with
9              {                       // random numbers
10                 Long n = (int)(java.Lang.Math.random()*199);
11                 arr.insert(n);
12             }
13         arr.display();               // display unsorted array
14
15         Long pivot = arr.getElement(0); // pivot value
16         System.out.print("Pivot is " + pivot);
17         int size = arr.size();
18
19         // partition array
20         int partDex = arr.partitionIt(0, size-1, pivot);
21         System.out.println(" with " + arr.getComparisonCount() + " comparisons and " + arr.getSwapCount() + " swaps.");
22         System.out.println(", Partition is at index " + partDex);
23         averageComparisonsIn100Runs += arr.getComparisonCount()/16.0;
24         arr.display();               // display partitioned array
25     }
26     System.out.println("Average number of comparisons in 100 runs: " + averageComparisonsIn100Runs/100 + " comparison/element");
27 } // end class PartitionApp
```

2 Merge Sort



```
1  public static void sort(int[] array, int leftIndex, int rightIndex, int n) {
2      if (n <= 1 || leftIndex >= rightIndex) {
3          return;
4      }
5      int midIndex = leftIndex + (rightIndex - leftIndex) / 2;
6      sort(array, leftIndex, midIndex, midIndex + 1);
7      sort(array, midIndex + 1, rightIndex, rightIndex - midIndex);
8      merge(array, leftIndex, midIndex, rightIndex);
9
10 }
```



```
1  private static void merge(int[] array, int left, int mid, int right) {
2      int leftSize = mid - left + 1;
3      int rightSize = right - mid;
4      int[] leftArray = new int[leftSize];
5      int[] rightArray = new int[rightSize];
6      for (int i = 0; i < leftSize; i++) {
7          leftArray[i] = array[left + i];
8          copies++;
9      }//copy data to temp arrays
10     for (int j = 0; j < rightSize; j++) {
11         rightArray[j] = array[mid + 1 + j];
12         copies++;
13     }
14     int i = 0, j = 0;
15     int k = left;
16     while (i < leftSize && j < rightSize) {
17         comparison++;
18         if (leftArray[i] <= rightArray[j]) {
19             array[k] = leftArray[i];//corrected from rightArray to leftArray
20             i++;
21             copies++;
22         } else {
23             array[k] = rightArray[j];//corrected from leftArray to rightArray
24             j++;
25             copies++;
26         }//merge the temp arrays
27         k++;
28     }
29
30     while (i < leftSize) {
31         array[k] = leftArray[i];
32         i++;
33         k++;
34         copies++;
35     }//copy remaining elements
36     while (j < rightSize) {
37         array[k] = rightArray[j];
38         j++;
39         k++;
40         copies++;
41     }
42 }
```

3 Shell Sort

```
1 public static void sort(int[] array) {
2     int h = 1;
3     int temp, i, j;
4     int n = array.Length;
5     while(h <= n / 3) {
6         h = h * 3 + 1; //Using Knuth's sequence to determine initial gap
7     }
8     while (h > 0) {
9         for(i = h; i < n; i++) {
10             temp = array[i];
11             copies++;
12             j = i;
13             while (j >= h && array[j - h] > temp) {
14                 array[j] = array[j - h];
15                 j -= h;
16                 copies++;
17                 comparison++;
18             }
19             array[j] = temp;
20             copies++;
21             if (j != i) {
22                 swaps++;
23             }
24         }
25         h = (h - 1) / 3; //Reduce the gap for the next pass
26     }
27 }
```

4 Quick Sort



```
1  public static void quickSort(int a[], int l, int r){
2      int p = a[(l+r)/2];
3      copies++;
4      int i = l, j = r;
5      while (i < j){
6          while (a[i] < p){// shift left pointer right if element is less than pivot
7              i++;
8              comparison++;
9          }
10         while (a[j] > p){// shift right pointer left if element is greater than pivot
11             j--;
12             comparison++;
13         }
14         comparison++;
15         if (i <= j){
16             int temp = a[i];
17             a[i] = a[j];
18             a[j] = temp;
19             copies += 3;
20             swaps++;
21             i++;
22             j--;
23         }
24         comparison++;
25     }
26     comparison++;
27     if (i < r){
28         quickSort(a, i, r);// sort right half
29     }
30     comparison++;
31     if (l < j){
32         quickSort(a, l, j);// sort left half
33     }
34 }
```

Table for analyzing:

COPIES/COMPARISONS/SWAPS			
	Merge Sort	Shell Sort	Quick Sort
10000	267232/120408/0	312353/161875/49718	170963/190707/52378
15000	417232/189266/0	524361/283883/78820	271416/281647/83471
20000	574464/260857/0	747739/417261/109748	372464/405189/114788
25000	734464/334069/0	923791/503313/140213	483572/503806/149056
30000	894464/408774/0	1138810/627381/171385	587429/624249/181513
35000	1058928/484356/0	1358798/747369/202412	699025/736104/216084
40000	1228928/561616/0	1635252/923823/236450	800292/894034/247940
45000	1398928/639592/0	1815185/1003756/269026	922667/949245/286286
50000	1568928/718172/0	2213452/1302023/306578	1040596/1092654/322439

This table shows the number of COPIES/COMPARISONS/SWAPS with respect to number of elements in the arrays. As illustrated on the table. Comparing to the simple sort table previously, it is inferred that the advanced sorts take fewer operations than the basic one.

5 Problem 5

————— *This is the end of the report* —————