Objective(s):

a. To implement a sorting algorithm as specified.
b. To measure its performance.

**Task 1:**

| 0 to blockSize -1 | | blockSize to 2 * blockSize - 1 | | 2blockSize to 3 * blockSize - 1 | | 3blockSize to 4 * blockSize - 1 | | 4blockSize to 5 * blockSize - 1 | | 5blockSize to arr.length - 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 13 | 9 | 15 | 18 | 21 | 13 | 8 | 5 | 11 | 3 |
| 10 | 13 | 9 | 15 | 18 | 21 | 8 | 13 | 5 | 11 | 3 |
| 9 | 10 | 13 | 15 | 8 | 13 | 18 | 21 | 3 | 5 | 11 |
| 8 | 9 | 10 | 13 | 13 | 15 | 18 | 21 | 3 | 5 | 11 |
| 3 | 5 | 8 | 9 | 10 | 11 | 13 | 13 | 15 | 18 | 21 |

The figure above shows the process of sorting n numbers. This algorithm process is as follows:

a. Break the data into chunks of blockSize i.e., if the blockSize is 32, blocks are as follows: 0 – 31, 32 – 63, …, (n – 2) * blockSize – (n – 1) * blockSize – 1, lastBlock. Keep in mind that lastBlock may not be full.
b. For each block, sort it.
c. Repeat

   Keep merging 2 consecutive blocks through all blocks. After each merge, the merged block's size is double, and its data is sorted.

   Until there is only one block left.

Complete the code given.

```
private static void whatSortIsThis(int [] arr) {
  int BLOCK_SIZE = arr.length / 4 > 32 ? 32 : arr.length / 4;
  for (int start = 0; start < arr.length; start += BLOCK_SIZE) {
      int end = Math.min(start + BLOCK_SIZE - 1, arr.length - 1);
      bite_size_sort(arr, start, end);
  }

  for (int mergeSize = BLOCK_SIZE; mergeSize < arr.length; mergeSize *= 2) {
    for (int left = 0; left < arr.length; left += 2 * mergeSize) {
        int mid = left + mergeSize - 1;
        int right = Math.min(0,0 /* your code1 */);
        if (mid < right)
            merge(arr,left, mid, right);
    }
  }
  System.out.println(Arrays.toString(arr));
}
private static void bite_size_sort(int [] b, int start, int end) {
  for (int i = 0 /* your code2 */; i < end; i++) {
      int j = i;
      int tmp = b[j];
      while (j > start && b[j - 1] > tmp) {
          b[j] = b[j-1];
          j--;
      }
      b[j] = tmp;
  }
}
private static void merge(int [] twob, int low, int mid, int high) {
    int [] leftArr = new int[mid - low + 1];
    int [] rightArr = new int[high - mid];
    System.arraycopy(twob, low, leftArr, 0, leftArr.length);
    System.arraycopy(twob, mid + 1, rightArr, 0, rightArr.length);

    int leftCounter = 0;
    int rightCounter = 0;
    int twobCounter = low;
    while (leftCounter < leftArr.length && rightCounter < rightArr.length) {
        twob[twobCounter++] = leftArr[leftCounter] < rightArr[rightCounter]
                        ? /* your code3 */;
    }
    while (leftCounter < leftArr.length)
        twob[twobCounter++] = leftArr[leftCounter++];
    while (rightCounter < rightArr.length)
        twob[twobCounter++] = rightArr[rightCounter++];
}
```

You may double check that the `println(Arrays.toString(arr));` in `whatSortIsThis()` produces the same output in main (i.e. array reference in main and arr in whatSortIsThis refers to the same array.

**Task 2:**

Use the below code to test whatSortIsThis performance. Arrays of 2,000,000 values random by shuffle are created. After each call, its elapse time is stored in is corresponded time array.

```
private static void testRuntime() {
  int ARRAY_SIZE = 2_000_000;
  int [] arr32 = new int[1];
  int [] arr2048 = new int[1];;
  int [] arr3 = new int[1];;
  int numIter = 20;
  int [] size32Time = new int[numIter];
  int [] size2048Time = new int[numIter];
  int [] sizeSortTime = new int[numIter];
  ArrayList<Integer> list = new ArrayList<>();
  for (int i = 1; i <= ARRAY_SIZE; i++)
      list.add(i);

  for (int i = 0; i < numIter; i++) {
      Collections.shuffle(list);

      arr32 = list.stream().mapToInt(Integer::intValue).toArray();
      arr2048 = list.stream().mapToInt(Integer::intValue).toArray();
      arr3 = list.stream().mapToInt(Integer::intValue).toArray();

      long startElapse = System.currentTimeMillis();
      whatSortIsThis(arr32, 32);
      size32Time[i] = (int)(System.currentTimeMillis() - startElapse);

      startElapse = System.currentTimeMillis();
      whatSortIsThis(arr2048, 2048);
      size2048Time[i] = (int)(System.currentTimeMillis() - startElapse);

      startElapse = System.currentTimeMillis();
      Arrays.sort(arr3);
      sizeSortTime[i] = (int)(System.currentTimeMillis() - startElapse);
  }
  System.out.println("confirm isSort " + isSort(arr32)
                     + " " + isSort(arr2048) + " " + isSort(arr3));
  System.out.println("takes " + Arrays.toString(size32Time));
  System.out.println("takes " + Arrays.toString(size2048Time));
  System.out.println("takes " + Arrays.toString(sizeSortTime));
}
private static boolean isSort(int [] arr) {
  for (int i = 1; i < arr.length; i++)
      if (arr[i - 1] > arr[i])
           return false;
  return true;
}
```

01286222 Lab 78b Name…… id ……

Instructions

1. capture your code

1.1 (your code 1)

1.2 (your code 2)

1.3 (your code 3)

2. Change numIter to 10. Capture your output.

3. What a brief opinion on which and why algorithm produced least elapse time outperforms the others.

**Submission:** This pdf

Due date: TBA

1.1)

```java
public static void whatSortIsThis(int [] arr, int PREFERRED_SIZE) {
    int BLOCK_SIZE = arr.length / 4 > PREFERRED_SIZE ? PREFERRED_SIZE : arr.length / 4;

    for (int start = 0; start < arr.length; start += BLOCK_SIZE) {
        int end = Math.min(start + BLOCK_SIZE - 1, arr.length - 1);
        bite_size_sort(arr, start, end);
    }
    for (int mergeSize = BLOCK_SIZE; mergeSize < arr.length; mergeSize *= 2) {
        for (int left = 0; left < arr.length; left += 2 * mergeSize) {
            int mid = left + mergeSize - 1;
            int right = Math.min(left + 2 * mergeSize - 1, arr.length - 1);
            if (mid < right)
            merge(arr,left, mid, right);
        }
    }
    System.out.println(Arrays.toString(arr));
}
```

1.2)

```java
private static void bite_size_sort(int [] b, int start, int end) {

    for (int i = start + 1; i <= end; i++) {
        int j = i;
        int tmp = b[j];
        while (j > start && b[j - 1] > tmp) {
            b[j] = b[j-1];
            j--;
        }
        b[j] = tmp ;
    }

}
```

1.3)

```
private static void merge(int [] twob, int low, int mid, int high) {
    int [] leftArr = new int[mid - low + 1];
    int [] rightArr = new int[high - mid];
    System.arraycopy(twob, low, leftArr, destPos:0, leftArr.length);
    System.arraycopy(twob, mid + 1, rightArr, destPos:0, rightArr.length);
    int leftCounter = 0;
    int rightCounter = 0;
    int twobCounter = low;
    while (leftCounter < leftArr.length && rightCounter < rightArr.length) {
        twob[twobCounter++] = leftArr[leftCounter] < rightArr[rightCounter]
        ? leftArr[leftCounter++]: rightArr[rightCounter++];
    }
    while (leftCounter < leftArr.length)
        twob[twobCounter++] = leftArr[leftCounter++];
    while (rightCounter < rightArr.length)
        twob[twobCounter++] = rightArr[rightCounter++];
}
```

2) num iter changed to 10

```
[7, 3, 1, 9, 6, 8, 4, 2, 5]
confirm isSort true
Array size :200000
whatSort with 32 blocks takes    : [42, 30, 18, 26, 19, 20, 21, 18, 19, 18]
whatSort with 2048 blocks takes : [56, 55, 55, 55, 54, 57, 55, 50, 53, 54]
built in array sort takes        : [82, 55, 12, 11, 12, 12, 11, 9, 10, 11]
```

3)

From the recorded results (I've ran this a few times), bult in array sort algorithm seems to be the fastest among the three. The whatSort algorithm with less block size seems to be faster than the one with bigger block size. The smaller one might have more operations on splitting and merging the blocks, but the operations inside the blocks seems to be faster than the bigger one. Lastly, I think the built-in sort is the fastest due to it being well optimized by the library developer and the each sorting algorithms may depend on the data size of the array.