# Lecture 10: Sorting

## CS 62

Spring 2019

William Devanny & Alexandra Papoutsaki

# Assignment 3

- What to do when you want to sort data that cannot fit in memory of your computer?
  - On-disk sorting

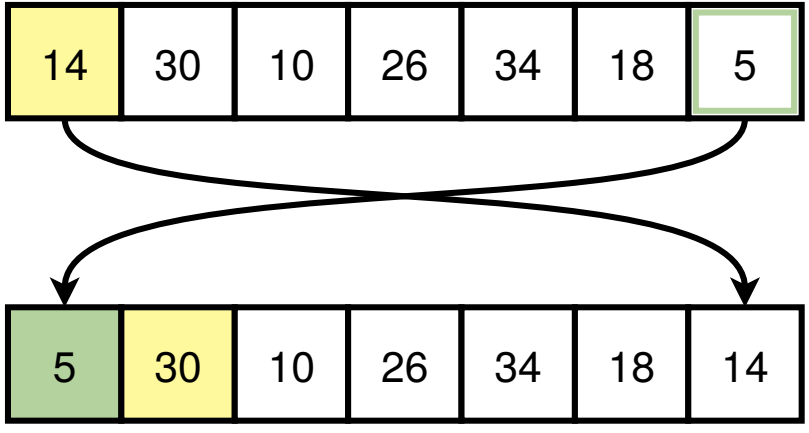
- Break data into chunks that will fit in memory, sort chunks, copy into new files: `0.tempfile`, `1.tempfile` , …
- Keep `ArrayList` of files
- Merge files together until one big sorted file.
- Note: You can’t keep file open as both read and write!

# Assignment 3 and Lab 3

- Read info on File I/O in Java and file systems in appendix to assignment.
- See on-line Streams cheat sheet

- Lab 3: More complexity/timing (sorting)

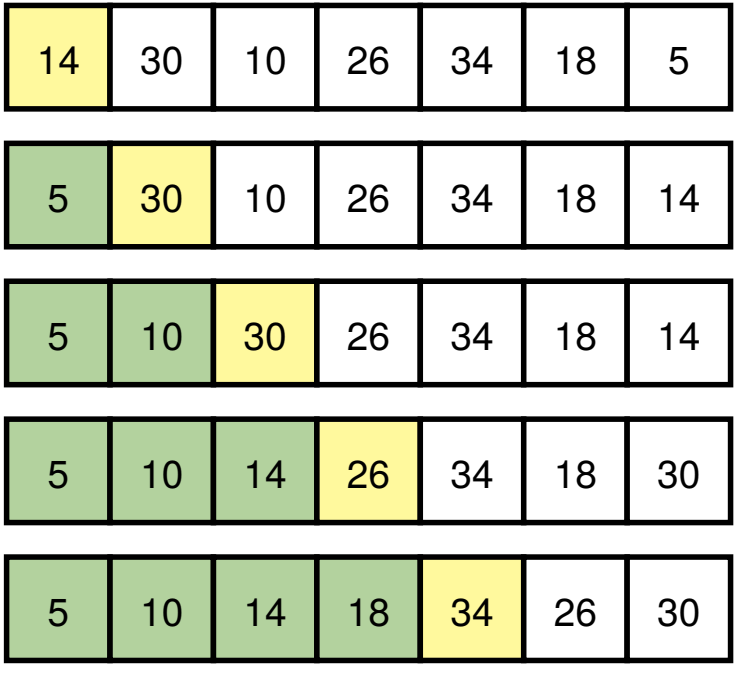
# Selection Sort




Array can be seen as split in two parts:
Unsorted (white) and sorted (green)

1. Select the first element of the unsorted list (yellow)
2. Find the smallest element in the remaining unsorted array (white)
3. Swap it with the first selected element and adjust the sorted array
4. Repeat with the rest of the array

# Selection Sort



# Selection Sort (helper)

```
/*
* @param array array of integers
* @param startIndex valid index into array
* @return index of smallest value in array[startIndex…length]
*/
protected static int indexofSmallest(int[] array, int startIndex) {
  int smallest = startIndex;
  for (int i = startIndex + 1; i < array.length; i++) {
    if (array[i] < array[smallest ]) {
      smallest = i;
    }
  }
  return smallest;
}
```

# Selection Sort (helper)

```
/*
* Swaps array[n1] and array[n2]
* @param array array of integers
* @param n1 valid index into array
* @param n2 valid index into array
* @throws IllegalArgumentException if n1 or n2 are not valid indices
*/
protected static void swap(int[] array, int n1, int n2) {
  if (n1 >= array.length || n2>=array.length || n1<0 || n2>0) {
    throw new IllegalArgumentException("Invalid array indices");
    }
  }
  int tmp = array[n1];
  array[n1] = array[n2];
  array[n2] = tmp;
}
```

# Recursive Selection Sort

```
/**
 * @param array array of integers
 * @param startIndex a valid index into array
 */
private static void selectionSortRecursive(int[] array, int startIndex) {
        if(startIndex > 0) {
                //recursively sort array[0,…startIndex-1];
                selectionSortRecursive(array, startIndex-1);
        }
        // find smallest element in array[startIndex…n]
        int smallest = indexOfSmallest(array, startIndex);
        // move smallest element to position startIndex
        swap(array, smallest, startIndex);
}
```

# Iterative Selection Sort

```
/**
* Sorts integer array using iterative selection sort
* @param array array of integers to be sorted
*/
private static void selectionSortIterative(int[] array) {
      for(int i = 0; i < array.length; ++i) {
            int min = indexOfSmallest(array, i);
            swap(array, i, min);
      }
}
```

# Time Complexity Analysis

- Count number of operations, i.e. comparisons of elements from array
- `indexOfSmallest(array,0)` → n-1 comparisons.
- `indexOfSmallest(array,1)` → n-2 comparisons.
- …
- `indexOfSmallest(array,n-1)` → 1 comparisons.

- In total: $(n-1) + \cdots + 2 + 1 = n(n-1)/2$
- If array has length $n$ then **`selectionSortRecursive(array,n)`** takes time $n(n-1)/2$, so $O(n^2)$

# Merge Sort

- Example of Divide & Conquer algorithm
  - Divide array in half
  - Sort each half
  - Merge halves together into completely sorted array

- *Needs extra space (not in-place)*
- Stable: two objects with equal keys appear in the same order in **sorted** output as they appear in the input unsorted array.

# MergeSort

```
/**
* MergeSort   Sorts data >= low and < high
* @param list data to be sorted
* @param low start of the data to be sorted
* @param high end of the data to be sorted (exclusive)
*/
private void mergeSort(int[] data, int low, int high){
   if( high-low > 1 ){
      int mid = low + (high-low)/2;
      mergeSort(data, low, mid);
      mergeSort(data, mid, high);
      merge(data, low, mid, high);
    }
}
```

```java
/** Merge data >= low and < high into sorted data.
* Data >= low and < mid are in sorted order.
* Data >= mid and < high are also in sorted order
*/
public void merge(int[] data, int low, int mid, int high){
    int[] temp = new int[high-low]; // make temporary array temp of size high-low
    int k = 0, i = low, j = mid;
    while( i < mid && j < high ){
        if( data[i] <= data[j]){
            temp[k] = data[i];
            i++;
        }else{
            temp[k] = data[j];
            j++;
        }
        k++;
    }
    // copy over the remaining data on the low to mid side if there is some remaining.
    // copy over the remaining data on the mid to high side if there is some remaining.
    // Only one of these two while loops should actually execute
    // copy the data back from temp to array
...
```

```
    // copy over the remaining data on the low to mid side if there is some remaining.
    while(i < mid){
            temp[k] = data[i];
             k++;
            i++;
    }
    // copy over the remaining data on the mid to high side if there is some remaining.
    while(j < high){
            temp[k] = data[j];
            k++;
            j++;
    }
    // Only one of these two while loops should actually execute

    // copy the data back from temp to array
    for(int index = 0; index < temp.length; index++ ){
            data[index+low]=temp[index];
    }
}
```

# Example

Sort: 85 24 63 47 17 31 96 50 (whiteboard)

# Complexity

- Claim: $mergeSort$ is $O(n \log n)$
  - where log is base 2
- Merge of two lists of combined size $n$ takes $\leq n - 1$ comparisons.
  - Think of merging [1,3,5,7] and [2,4,6,8]
- If $l$ levels:
  - $n/2^l$=1
  - $n = 2^l$
  - $l = \log n$

- $\log n$ levels
- each taking $O(n)$ operations
- $O(n \log n)$ in total