Sorting

CSE260, Computer Science B: Honors
Stony Brook University

http://www.cs.stonybrook.edu/~cse260
Objectives

- To study and analyze time complexity of various sorting algorithms
  - To design, implement, and analyze insertion sort
  - To design, implement, and analyze bubble sort
  - To design, implement, and analyze merge sort
  - To design, implement, and analyze quick sort
  - To design and implement a binary heap
  - To design, implement, and analyze heap sort
  - To design, implement, and analyze bucket sort and radix sort
  - To design, implement, and analyze external sort for files that have a large amount of data
Why study sorting?

• Sorting is a classic subject in computer science
• Reasons for studying sorting algorithms:
  • Sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems
  • Sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays
  • Sorting algorithms are excellent examples to demonstrate algorithm performance
What data to sort?

- The data to be sorted might be integers, doubles, characters, or objects.
- The Java API contains several overloaded sort methods for sorting primitive type values and objects in the `java.util.Arrays` and `java.util.Collections` class.
- For simplicity, this section assumes:
  - data to be sorted are integers
  - data are sorted in ascending order
  - data are stored in an array
- The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in an `ArrayList` or a `LinkedList`. 
The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

```
int[] myList = {2, 9, 5, 4, 8, 1, 6};  // Unsorted
```

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step 2: The sorted sublist is \{2, 9\}. Insert 5 into the sublist.

Step 3: The sorted sublist is \{2, 5, 9\}. Insert 4 into the sublist.

Step 4: The sorted sublist is \{2, 4, 5, 9\}. Insert 8 into the sublist.

Step 5: The sorted sublist is \{2, 4, 5, 8, 9\}. Insert 1 into the sublist.

Step 6: The sorted sublist is \{1, 2, 4, 5, 8, 9\}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.
How to Insert?

Step 1: Save 4 to a temporary variable `currentElement`

Step 2: Move `list[2]` to `list[3]`

Step 3: Move `list[1]` to `list[2]`

Step 4: Assign `currentElement` to `list[1]`
From Idea to Solution

for (int i = 1; i < list.length; i++) {
    insert list[i] into a sorted sublist list[0..i-1] so that
    list[0..i] is sorted
}

Expand

int currentElement = list[i];
int k;
for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
    list[k + 1] = list[k];
}
// Insert the current element into list[k + 1]
list[k + 1] = currentElement;

time: $O(n^2)$
public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
        int currentElement = list[i];
        int k;
        for (k = i - 1; k >= 0 && list[k] > currentElement; k--)
            list[k + 1] = list[k];
        // Insert the current element into list[k + 1]
        list[k + 1] = currentElement;
    }
}

int[] list = {1, 9, 4, 6, 5, -4};
InsertionSort.insertionSort(list);

\[
T(n) = (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n - 1) + c) \\
= 2(1 + 2 + \cdots + n - 1) + c(n - 1) \\
= 2 \frac{(n - 1)n}{2} + cn - c = n^2 - n + cn - c \\
= O(n^2)
\]
Bubble Sort

- Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.
  - After the first pass, the last element becomes the largest in the array.
  - After the second pass, the second-to-last element becomes the second largest in the array.
- Bubble sort is also called sinking sort because the smaller values gradually “bubble” their way to the top and the larger values sink to the bottom.

```java
for (int k = 1; k < list.length; k++) {
    // Perform the kth pass
    for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1])
            // swap list[i] with list[i + 1];
            ...
    }
}
```
From Idea to Solution

- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted
  - If no swap takes place in a pass, there is no need to perform the next pass, because all the elements are already sorted
  - We can use this property to improve the previous algorithm

```java
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
    // Array may be sorted and next pass not needed
    needNextPass = false;
    // Perform the kth pass
    for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1]) {
            // swap list[i] with list[i + 1];
            needNextPass = true; // Next pass still needed
        }
    }
}
```
Bubble Sort

- Example:

  | 2 9 5 4 8 1 | 2 5 4 8 1 9 | 2 4 5 1 8 9 | 2 4 1 5 8 9 | 1 2 4 5 8 9 |
  | 2 5 9 4 8 1 | 2 4 5 8 1 9 | 2 4 5 1 8 9 | 2 1 4 5 8 9 |
  | 2 5 4 9 8 1 | 2 4 5 8 1 9 | 2 4 1 5 8 9 |
  | 2 5 4 8 9 1 | 2 4 5 1 8 9 |
  | 2 5 4 8 1 9 |

  (a) 1st pass  (b) 2nd pass  (c) 3rd pass  (d) 4th pass  (e) 5th pass

- In the best case, the bubble sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed. Since the number of comparisons is $n - 1$ in the first pass, the best-case time for a bubble sort is $O(n)$.

- **Worse case:** $(n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{n^2}{2} - \frac{n}{2}$

  time: $O(n^2)$
Merge Sort

- **Merge sort** is a divide and conquer algorithm that was invented by John von Neumann in 1945.
- Divide the unsorted list into $n$ sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining.
- This will be the sorted list.
Merge Sort

- **Divide**: Split the array into smaller parts.
- **Conquer**: Sort the smaller parts recursively.
- **Merge**: Combine the sorted parts to form the final sorted array.

Diagram:

```
2 9 5 4 8 1 6 7
  
2 9 5 4
  
2 9
  
2 9

5 4
  
5
  
4

8 1 6 7
  
8 1
  
8

6 7
  
6
  
6

merge

2 9 5 4 8 1 6 7
  merge

merge

2 4 5 9
  merge

merge

1 2 4 5 6 7 8 9
```
Merge Sort

mergeSort(list):
    firstHalf = mergeSort(firstHalf);
    secondHalf = mergeSort(secondHalf);
    list = merge(firstHalf, secondHalf);
public static void mergeSort(int[] list) {
    if (list.length > 1) {
        // Merge sort the first half
        int[] firstHalf = new int[list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
        mergeSort(firstHalf);

        // Merge sort the second half
        int secondHalfLength = list.length - list.length / 2;
        int[] secondHalf = new int[secondHalfLength];
        System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);
        mergeSort(secondHalf);

        // Merge firstHalf with secondHalf into list
        merge(firstHalf, secondHalf, list);
    }
}
/** Merge two sorted lists */
public static void merge(int[] list1, int[] list2, int[] temp) {
    int current1 = 0; // Current index in list1
    int current2 = 0; // Current index in list2
    int current3 = 0; // Current index in temp

    while (current1 < list1.length && current2 < list2.length) {
        if (list1[current1] < list2[current2])
            temp[current3++] = list1[current1++];
        else
            temp[current3++] = list2[current2++];
    }

    while (current1 < list1.length)
        temp[current3++] = list1[current1++];

    while (current2 < list2.length)
        temp[current3++] = list2[current2++];
}

public static void main(String[] args) {
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    mergeSort(list);
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
Merge Two Sorted Lists

(a) After moving 1 to temp
(b) After moving all the elements in list2 to temp
(c) After moving 9 to temp
Merge Sort Time

• Let $T(n)$ denote the time required for sorting an array of $n$ elements using merge sort.
• Without loss of generality, assume $n$ is a power of 2.
• The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays.

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \text{mergetime}$$

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n)$$
Merge Sort Time

- The first $T(n/2)$ is the time for sorting the first half of the array and the second $T(n/2)$ is the time for sorting the second half.
- To merge two subarrays, it takes at most $n-1$ comparisons to compare the elements from the two subarrays and $n$ moves to move elements to the temporary array.

\[
T(n) = 2T\left(\frac{n}{2}\right) + 2n - 1 = 2\left(2T\left(\frac{n}{4}\right) + 2\frac{n}{2} - 1\right) + 2n - 1 = 2^2 T\left(\frac{n}{2^2}\right) + 2n - 2 + 2n - 1
\]

\[
= 2^k T\left(\frac{n}{2^k}\right) + 2n - 2^{k-1} + ... + 2n - 2 + 2n - 1
\]

\[
= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + 2n - 2^{\log n-1} + ... + 2n - 2 + 2n - 1
\]

\[
= n + 2n \log n - 2^{\log n} + 1 = 2n \log n + 1 = O(n \log n)
\]
Quick Sort

• **Quick sort**, developed by C. A. R. Hoare (1962), works as follows:
  • The algorithm selects an element, called the **pivot**, in the array
  • **Divide** the array into two parts such that all the elements in the first part are **less** than or equal to the pivot and all the elements in the second part are **greater** than the pivot
  • Recursively apply the quick sort algorithm to the first part and then the second part
Quick Sort

(a) The original array

(b) The original array is partitioned

(c) The partial array (4 2 1 3 0) is partitioned

(d) The partial array (0 2 1 3) is partitioned

(e) The partial array (2 1 3) is partitioned
Partition with forward and backward search

(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

The index of the pivot is returned
public static void quickSort(int[] list) {
    quickSort(list, 0, list.length - 1);
}

different version:

public static void quickSort(int[] list, int first, int last) {
    if (last > first) {
        int pivotIndex = partition(list, first, last);
        quickSort(list, first, pivotIndex - 1);
        quickSort(list, pivotIndex + 1, last);
    }
}

different version:

public static int partition(int[] list, int first, int last) {
    int pivot = list[first]; // Choose the first element as pivot
    int low = first + 1; // Index for forward search
    int high = last; // Index for backward search
    while (high > low) {
        // Search forward from left
        while (low <= high && list[low] <= pivot)
            low++;
        // Search backward from right
        while (low <= high && list[high] > pivot)
            high--;
    }
    return pivotIndex;
}
// Swap two elements in the list
if (high > low) {
    int temp = list[high];
    list[high] = list[low];
    list[low] = temp;
}

// Account for duplicated elements:
while (high > first && list[high] == pivot)
    high--;

// Swap pivot with list[high]
if (pivot > list[high]) {
    list[first] = list[high];
    list[high] = pivot;
    return high;
} else {
    return first;
}
}
Quick Sort Time

- To **partition** an array of $n$ elements, it takes $n-1$ comparisons and $n$ moves in the worst case.
- So, the time required for partition is $O(n)$. 
Worst-Case Time

- In the worst case, each time the pivot divides the array into one big subarray with the other empty.
- The size of the big subarray is one less than the one before divided.
- The algorithm requires:

\[(n - 1) + (n - 2) + ... + 2 + 1 = O(n^2)\]
Best-Case Time

• In the best case, each time the pivot divides the array into two parts of about the same size
• Let \( T(n) \) denote the time required for sorting an array of \( n \) elements using quick sort. So,

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = O(n \log n)
\]
Average-Case Time

- On the average, each time the pivot will not divide the array into two parts of the same size nor one empty part.
- Statistically, the sizes of the two parts are very close => the average time is also \( O(n \log n) \).
- Both merge sort and quick sort employ the divide-and-conquer approach.
  - For merge sort, the bulk of the work is to merge two sublists.
  - Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case.
  - Merge sort requires a temporary array for sorting two subarrays.
  - Quick sort does not need additional array space. Thus, quick sort is more space efficient than merge sort.
Heap

- **Heap sort** uses a binary heap: it first adds all the elements to a heap and then removes the largest elements successively to obtain a sorted list.
  - Heap is a useful data structure for designing efficient sorting algorithms and priority queues.
- A **binary tree** is a hierarchical structure: it either is empty or it consists of an element, called the **root**, and two distinct binary trees, called the **left subtree** and **right subtree**.
  - The **length** of a path is the number of the edges in the path.
  - The **depth** of a node is the length of the path from the root to that node.
- A **binary heap** is a binary tree with the following properties:
  - It is a complete binary tree, and
  - Each node is greater than or equal to any of its children.
Complete Binary Tree

- A binary tree is **complete** if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most.

- For example, in the following figure, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete.

  - Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).
Storing a Heap

- A heap can be stored in an **ArrayList** or an array if the heap size is known in advance
- For a node at position \( i \), its left child is at position \( 2i+1 \) and its right child is at position \( 2i+2 \), and its parent is at index \( (i-1)/2 \)
  - For example: the root is at position 0, and its two children are at positions 1 and 2
  - The node for element 39 is at position 4, so its left child (element 14) is at 9 (\( 2\times4+1 \)), its right child (element 33) is at 10 (\( 2\times4+2 \)), and its parent (element 42) is at 1 (\( (4-1)/2 \)).
Adding Elements to the Heap

• To add a new node to the heap, first add it to the end of the heap and then rebuild the tree with this algorithm:

Let the last node be the current node;
while (the current node is greater than its parent) {
    Swap the current node with its parent;
    Now the current node is one level up;
}
Adding Elements to the Heap

- Suppose a heap is initially empty. After adding numbers 3, 5, 1, 19, 11, and 22 in this order:

(a) After adding 3
(b) After adding 5
(c) After adding 1
(d) After adding 19
(e) After adding 11
(f) After adding 22
Adding Elements to the Heap

• Adding 88 into the heap:
  • Place the new node 88 at the end of the tree
  • Swap 88 with 19
  • Swap 88 with 22
Removing the Root and Rebuild the Tree

- Often you need to remove the maximum element, which is the root in a heap
- After the root is removed, the tree must be rebuilt to maintain the heap property using this algorithm:

Move the last node to replace the root;
Let the root be the current node;
while (the current node has children and the current node is smaller than one of its children) {
    Swap the current node with the larger of its children;
    Now the current node is one level down;
}
Removing the Root and Rebuild the Tree

- Removing root 62 from the heap
Removing the Root and Rebuild the Tree

Move 9 to root

```
  9
 /   \
42    59
/     /
32    44
|      |
22    13
|      |
29    17
```

37
Removing the Root and Rebuild the Tree

Swap 9 with 59
Removing the Root and Rebuild the Tree

Swap 9 with 44
Removing the Root and Rebuild the Tree
The Heap Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list: java.util.ArrayList&lt;E&gt;</td>
<td></td>
</tr>
<tr>
<td>+Heap()</td>
<td>Creates a default empty heap.</td>
</tr>
<tr>
<td>+Heap(objects: E[])</td>
<td>Creates a heap with the specified objects.</td>
</tr>
<tr>
<td>+add(newObject: E): void</td>
<td>Adds a new object to the heap.</td>
</tr>
<tr>
<td>+remove(): E</td>
<td>Removes the root from the heap and returns it.</td>
</tr>
<tr>
<td>+getSize(): int</td>
<td>Returns the size of the heap.</td>
</tr>
</tbody>
</table>
public class Heap<E extends Comparable> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

    /** Create a default heap */
    public Heap() {
    }

    /** Create a heap from an array of objects */
    public Heap(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    /** Add a new object into the heap */
    public void add(E newObject) {
        list.add(newObject); // Append to the heap
        int currentIndex = list.size() - 1; // The index of the last node
        while (currentIndex > 0) {
            int parentIndex = (currentIndex-1)/2;
            // Swap if the current object is greater than its parent
            if (list.get(currentIndex).compareTo(list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex, list.get(parentIndex));
                list.set(parentIndex, temp);
            } else break; // the tree is a heap now
        }
    }
}
currentIndex = parentIndex;
}
}

/** Remove the root from the heap */
public E remove() {
    if (list.size() == 0) return null;

    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);

    int currentIndex = 0;
    while (currentIndex < list.size()) {
        int leftChildIndex = 2 * currentIndex + 1;
        int rightChildIndex = 2 * currentIndex + 2;

        // Find the maximum between two children
        if (leftChildIndex >= list.size()) break;  // The tree is a heap
        int maxIndex = leftChildIndex;
        if (rightChildIndex < list.size()) {
            if (list.get(maxIndex).compareTo(list.get(rightChildIndex)) < 0) {
                maxIndex = rightChildIndex;
            }
        }
    
    return removedObject;
}
// Swap if the current node is less than the maximum
if (list.get(currentIndex).compareTo(list.get(maxIndex)) < 0) {
    E temp = list.get(maxIndex);
    list.set(maxIndex, list.get(currentIndex));
    list.set(currentIndex, temp);
    currentIndex = maxIndex;
} else
    break; // The tree is a heap

return removedObject;

/** Get the number of nodes in the tree */
public int getSize() {
    return list.size();
}
Heap Sort

public class HeapSort {
    public static <E extends Comparable> void heapSort(E[] list) {
        // Create a Heap of integers
        Heap<E> heap = new Heap<E>();

        // Add elements to the heap
        for (int i = 0; i < list.length; i++)
            heap.add(list[i]);

        // Remove elements from the heap
        for (int i = list.length - 1; i >= 0; i--)
            list[i] = heap.remove();
    }
}

/** A test method */
public static void main(String[] args) {
    Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    heapSort(list);
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
}

Heap Sort Time

- Let $h$ denote the height for a heap of $n$ elements. Since a heap is a complete binary tree, the first level has 1 node, the second level has 2 nodes, the $k$th level has $2^{(k-1)}$ nodes, the $(h-1)$th level has $2^{(h-2)}$ nodes, and the $h$th level has at least one node and at most $2^{(h-1)}$ nodes. Therefore,

\[
1 + 2 + ... + 2^{h-2} < n \leq 1 + 2 + ... + 2^{h-2} + 2^{h-1}
\]

\[
2^{h-1} - 1 < n \leq 2^h - 1
\]

\[
2^{h-1} < n + 1 \leq 2^h
\]

\[
\log 2^{h-1} < \log(n + 1) \leq \log 2^h
\]

\[
h - 1 < \log(n + 1) \leq h
\]

- Thus, $\log(n + 1) \leq h < \log(n + 1) + 1$

- Hence, the height of the heap is $O(\log n)$
Heap Sort Time

- Since the add method traces a path from a leaf to a root, it takes at most $h$ steps to add a new element to the heap.
- Thus, the total time for constructing an initial heap is $O(n \log n)$ for an array of $n$ elements.
- Since the remove method traces a path from a root to a leaf, it takes at most $h$ steps to rebuild a heap after removing the root from the heap.
- Since the remove method is invoked $n$ times, the total time for producing a sorted array from a heap is $O(n \log n)$.
- Merge sort requires a temporary array for merging two subarrays; a heap sort does not need additional array space.
- Therefore, a heap sort is more space efficient than a merge sort.
Bucket Sort and Radix Sort

• All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable objects)

• These algorithms sort the elements by comparing their keys

• The lower bound for general sorting algorithms is $O(n \log n)$

• So, no sorting algorithms based on comparisons can perform better than $O(n \log n)$

• However, if the keys are small integers, you can use bucket sort without having to compare the keys
Bucket Sort

- The bucket sort algorithm works as follows
  - Assume the keys are in the range from 0 to $N-1$
  - We need $N$ buckets labeled 0, 1, ..., and $N-1$
  - If an element’s key is $i$, the element is put into the bucket $i$
  - Each bucket holds the elements with the same key value

- You can use an `ArrayList` to implement a bucket
void bucketSort(E[] list) {
    E[] bucket = (E[]) new java.util.ArrayList[t+1];
    // Distribute the elements from list to buckets
    for (int i = 0; i < list.length; i++) {
        // Assume element has the getKey() method
        int key = list[i].getKey();
        if (bucket[key] == null)
            bucket[key] = new java.util.ArrayList<>();
        bucket[key].add(list[i]);
    }
    // Now move the elements from the buckets back to list
    int k = 0; // k is an index for list
    for (int i = 0; i < bucket.length; i++) {
        if (bucket[i] != null) {
            for (int j = 0; j < bucket[i].size(); j++)
                list[k++] = bucket[i].get(j);
        }
    }
}
Bucket Sort

- Takes $O(n + t)$ time to sort the list and uses $O(n + t)$ space, where $n$ is the list size.
- Note that if $t$ is too large, using the bucket sort is not desirable.
  - Instead, you can use a radix sort.
  - The radix sort is based on the bucket sort, but a radix sort uses only ten buckets.
- Bucket sort is stable, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list.
  - That is, if element $e_1$ and element $e_2$ have the same key and $e_1$ precedes $e_2$ in the original list, $e_1$ still precedes $e_2$ in the sorted list.
Radix Sort

- Assume that the keys are positive integers
- The idea for the radix sort is to divide the keys into subgroups based on their radix positions
- It applies a bucket sort repeatedly for the key values on radix positions, starting from the least-significant position

Radix sort takes $O(d*n)$ time to sort $n$ elements with integer keys, where $d$ is the maximum number of the radix positions among all keys.
Radix Sort

- Sort 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

- Remove elements from the buckets:
  230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 9

(c) Paul Fodor (CS Stony Brook) & Pearson
External Sort

- All the sort algorithms discussed in the preceding sections assume that all data to be sorted is available at one time in internal memory such as an array.
- To sort data stored in an external file, you may first bring data to the memory, then sort it internally.
- However, if the file is too large, all data in the file cannot be brought to memory at one time.
Phase I

- Repeatedly bring data from the file to an array, sort the array using an internal sorting algorithm, and output the data from the array to a temporary file.

![Diagram](c) Paul Fodor (CS Stony Brook) & Pearson
Phase II

- Merge a pair of sorted segments (e.g., S1 with S2, S3 with S4, ..., and so on) into a larger sorted segment and save the new segment into a new temporary file. Continue the same process until one sorted segment results.
Implementing Phase II

- Each merge step merges two sorted segments to form a new segment. The new segment doubles the number of elements. So the number of segments is reduced by half after each merge step. A segment is too large to be brought to an array in memory. To implement a merge step, copy half number of segments from file f1.dat to a temporary file f2.dat. Then merge the first remaining segment in f1.dat with the first segment in f2.dat into a temporary file named f3.dat.
### Implementing Phase II

<table>
<thead>
<tr>
<th>S₁</th>
<th>S₂</th>
<th>S₃</th>
<th>S₄</th>
<th>S₅</th>
<th>S₆</th>
<th>S₇</th>
<th>S₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1.dat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copy to f2.dat

<table>
<thead>
<tr>
<th>S₁</th>
<th>S₂</th>
<th>S₃</th>
<th>S₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2.dat</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S₁, S₅ merged</th>
<th>S₂, S₆ merged</th>
<th>S₃, S₇ merged</th>
<th>S₄, S₈ merged</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.dat</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>