CSE 143
Lecture 22

Merge Sort

slides created by Ethan Apter
http://www.cs.washington.edu/143/

Some Sorting Algorithms

- **Insertion sort:** add values to a sorted list so as to maintain sorted order
  - the "sorted list" can originally be empty so that you can sort many values
  - this is the sorting algorithm we used in **`SortedIntList`**

- **Selection sort:** scan an unsorted list for the smallest value, and move the smallest value to the front. Repeat this process for the remaining values until the whole list is sorted

- Both insertion sort and selection sort are O(n^2) algorithms

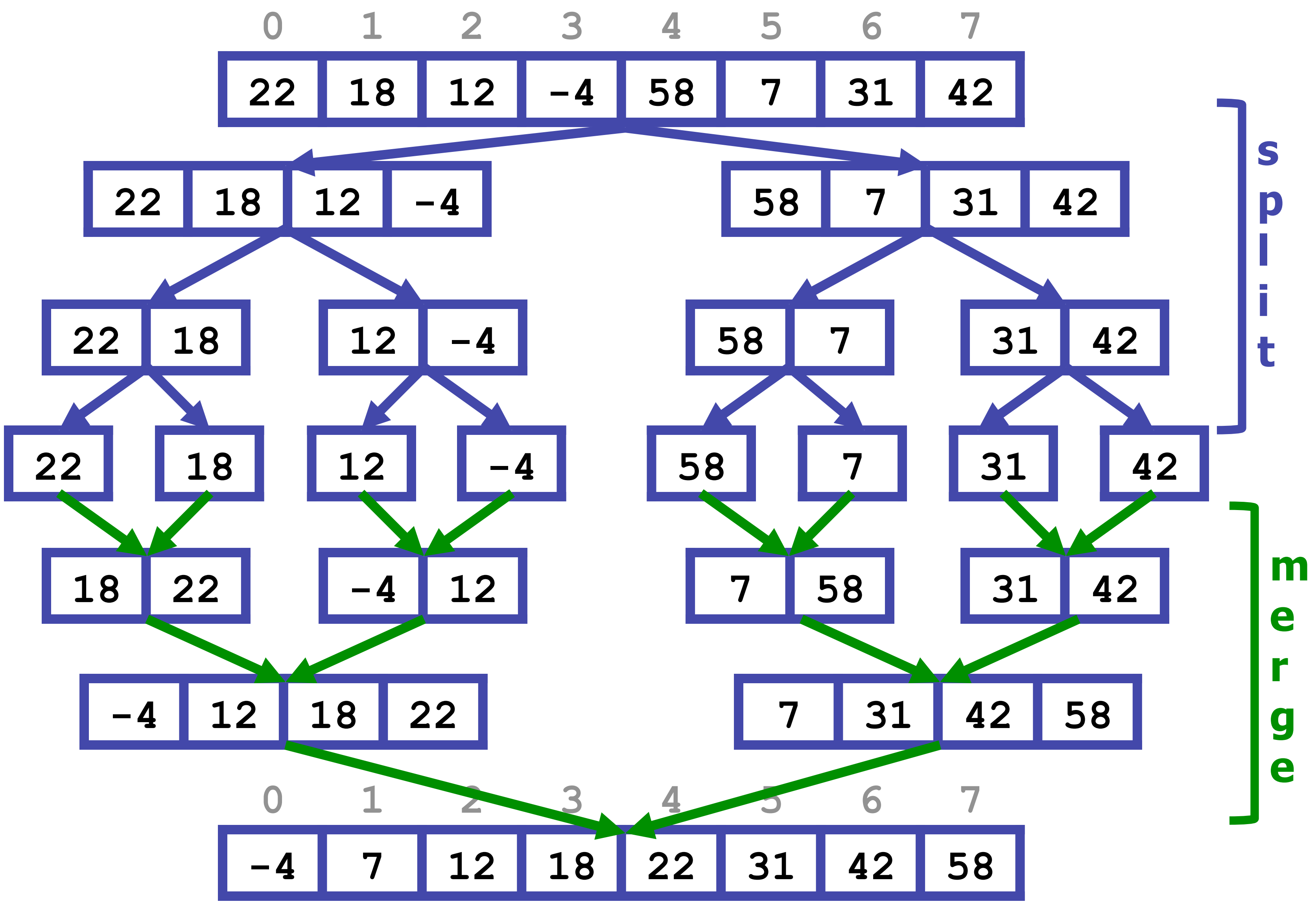
Merge Sort

- **Merge sort:**
  - divide a list into two halves
  - sort the halves
  - recombine the sorted halves into a sorted whole

- Merge sort is an example of a "divide and conquer" algorithm

- **divide and conquer algorithm:** an algorithm that repeatedly divides the given problem into smaller pieces that can be solved more easily
  - it's easier to sort the two small lists than the one big list

Merge Sort Picture



mergeInto

- **Merge sort:**
  - divide a list into two halves
  - sort the halves
  - recombine the sorted halves into a sorted whole

We're going to write this part first

- We're going to write the following method:

```
// post: copies the values from list1 and list2
//       into result so that the values in result
//       are in sorted order
private static void mergeInto(int[] result,
                              int[] list1,
                              int[] list2) {
    ...
}
```

mergeInto

- So how do we actually merge the two sorted lists into one sorted result list?

- One (wrong) way would just be to blindly copy the values from the two sorted lists into the result list and then sort the result list
  - this doesn't take advantage of the fact that the two lists are already sorted

- Instead, we'll repeatedly select the smallest value from both sorted lists and put this value into the result list
  - because the two sorted lists are sorted, we know that their smallest values are found at the front

`mergeInto`

- So to compare the smallest values, we'll do something like this

```
if (list1[0] <= list2[0])
    result[0] = list1[0];
else
    result[0] = list2[0];
```

- Obviously, this only handles the very first value
- We need to use a loop and update our indexes in order to get this working correctly
- But how many indexes do we need?

`mergeInto` Indexes

- We have three arrays, so we need three indexes
  - we need an index for list1, which tells us how many values from list1 we've copied so far
  - we need an index for list2, which tells us how many values from list2 we've copied so far
  - we need an index for result, which tells us how many values total we've copied from list1 and list2
    - we could just use the sum of the first two indexes for this index

- So we have the following indexes:

```
int i1 = 0; // index for list1
int i2 = 0; // index for list2
int i = 0;  // index for result (equals i1 + i2)
```

mergeInto

- So now we'll update our previous code to use a loop and our indexes

```
int i1 = 0;
int i2 = 0;
for (int i = 0; i < result.length; i++) {
    if (list1[i1] <= list2[i2]) {
        result[i] = list1[i1];
        i1++;
    } else {
        result[i] = list2[i2];
        i2++;
    }
}
```

But this doesn't quite work!

mergeInto

- Right now, our code always compares a value from list1 with a value from list2

- But, because we're coping a single value into result per loop iteration, we'll finish copying all the values from one of the sorted lists before the other

- So we also need to check if we've copied all the values from a list
  - if we've already copied all the values from list1, copy the value from list2
  - if we've already copied all the values from list2, copy the value from list1

mergeInto

- Updated **mergeInto** code:

```
int i1 = 0;
int i2 = 0;
for (int i = 0; i < result.length; i++) {
    if (i2 >= list2.length || (i1 < list1.length &&
                                    list1[i1] <= list2[i2])) {
        result[i] = list1[i1];
        i1++;
    } else {
        result[i] = list2[i2];
        i2++;
    }
}
```

This code is a little subtle. It relies on the short-circuiting property of && and ||

- This code is finished (it just needs to be put inside the method header)

`mergeInto` Final Code

```
private static void mergeInto(int[] result,
                              int[] list1,
                              int[] list2) {
  int i1 = 0;
  int i2 = 0;
  for (int i = 0; i < result.length; i++) {
    if (i2 >= list2.length || (i1 < list1.length &&
                               list1[i1] <= list2[i2])) {
      result[i] = list1[i1];
      i1++;
    } else {
      result[i] = list2[i2];
      i2++;
    }
  }
}
```

`mergeInto`'s Preconditions

- But notice our **`mergeInto`** method has some preconditions:
  - both **`list1`** and **`list2`** must already be sorted
  - **`result`**'s length must equal **`list1`**'s length plus **`list2`**'s length

- Because this is a private method, we're just going to settle for only commenting these preconditions

- Updated comment:

```
// pre:  list1 and list2 are sorted
//       result.length == list1.length + list2.length
// post: copies the values from list1 and list2
//       into result so that the values in result
//       are in sorted order
```

Merge Sort

- Recall that merge sort consists of the following steps:
  - divide a list into two halves
  - sort the halves
  - recombine the sorted halves into a sorted whole

- We've written the "recombine" step (**`mergeInto`**)
  - But now we have to write the first two steps

- Let's define our public sort method:

```
// post: sorts the given array into non-decreasing order
public static void sort(int[] list) {
    ...
}
```

Dividing the List in Half

- First we have to divide our list into two lists that are half the size of the original list

- To do this, we need to create a place to store these two lists
  - we'll create two new integer arrays

- A first attempt at creating the two new integer arrays:

```
int[] list1 = new int[list.length / 2];
int[] list2 = new int[list.length / 2];
```

- But what if the length of our array is odd?
  - e.g. if list.length == 1001, then list.length / 2 == 500
    - and we'll miss an element!
  - we need to handle the odd case

Dividing the List in Half

- Here’s some better code:

```
int[] list1 = new int[list.length / 2];
int[] list2 = new int[list.length - list.length / 2];
```

- And now we can also copy the first half of the list into **`list1`** and the second half of the list into **`list2`**

- Here’s the code for copying the values

```
for (int i = 0; i < list1.length; i++)
    list1[i] = list[i];
for (int i = 0; i < list2.length; i++)
    list2[i] = list[i + list1.length];
```

Merge Sort

- So here's our code for **`sort`** so far:

```
public static void sort(int[] list) {
    int[] list1 = new int[list.length / 2];
    int[] list2 = new int[list.length - list.length / 2];
    for (int i = 0; i < list1.length; i++)
        list1[i] = list[i];
    for (int i = 0; i < list2.length; i++)
        list2[i] = list[i + list1.length];
    ...
}
```

- So, we've successfully divided the list in half, and we've written code that can merge two sorted lists
  - so we still have to sort the two halves

Sorting the Halves

- But how will we sort the two halves?
  - if only we had a method that could sort a list...
  - oh wait! That's what we're writing!
- If we're going to make our sort method recursive, we need to modify it to have a base case
  - what's an easy list to sort?
    - the empty list (sorted by definition)
    - the list with one element (sorted by definition)
- So if our sort is called on lists with size <= 1, we don't have to do anything

Sorting the Halves

- Modified **`sort`** to have a base case:

```
public static void sort(int[] list) {
  if (list.length > 1) {
    int[] list1 = new int[list.length / 2];
    int[] list2 = new int[list.length - list.length / 2];
    for (int i = 0; i < list1.length; i++)
      list1[i] = list[i];
    for (int i = 0; i < list2.length; i++)
      list2[i] = list[i + list1.length];
    ...
  }
}
```

- Now we can just call **`sort`** on the two halves (**`list1`** and **`list2`**), and we'll eventually reach our base case

Sorting the Halves

- Modified **`sort`** that calls itself to sort the two halves:

```
public static void sort(int[] list) {
  if (list.length > 1) {
    int[] list1 = new int[list.length / 2];
    int[] list2 = new int[list.length - list.length / 2];
    for (int i = 0; i < list1.length; i++)
      list1[i] = list[i];
    for (int i = 0; i < list2.length; i++)
      list2[i] = list[i + list1.length];
    sort(list1);
    sort(list2);
    ...
  }
}
```

- Finally, we just merge the two halves into the original list

`sort` Final Code

- Final version of **`sort`**:

```
public static void sort(int[] list) {
  if (list.length > 1) {
    int[] list1 = new int[list.length / 2];
    int[] list2 = new int[list.length - list.length / 2];
    for (int i = 0; i < list1.length; i++)
      list1[i] = list[i];
    for (int i = 0; i < list2.length; i++)
      list2[i] = list[i + list1.length];
    sort(list1);
    sort(list2);
    mergeInto(list, list1, list2);
  }
}
```

Merge Sort is Stable

- Merge sort can be written so that it is a stable sort

- **stable sort:** a sorting algorithm that maintains the relative positions of equal elements
  - So, if there are multiple 3s in a list, a stable sort will put
    - put the 3 that occurred first before the others
    - put the 3 that occurred second after the first and before the others
    - ...
    - the 3 that occurred last after the others

- And the way we've written merge sort makes it stable
  - because of how we divided our list into halves in **`sort`**
  - and because of how we break ties during the comparison of values in **`mergeInto`**

Analyzing Our Performance

- Let's compare the performance of our merge sort to Sun's **`Arrays.sort`**

- Here's a sample run from **`IntArraySorter.java`** (on the website) run on a 2.53 GHz Intel Core 2 Duo Macbook

```
This program compares the performance of the
merge sort written in CSE 143 against the
performance of Sun's Arrays.sort method by
timing both methods on random, identical int
arrays of length 2500000

Sun's time:  0.507 seconds
Our time:    1.135 seconds
Ratio (Sun's/our): 2.24
```

Analyzing Our Performance

- Sun's sort is faster than ours
  - but only by a factor of about 2.5
- That's really good! It didn't take us long to write this, and Sun's **`Arrays.sort`** is a professional sorting method
- Sun has had years to fine-tune the performance of Arrays.sort, and yet we wrote a reasonably competitive merge sort in less than an hour!
- So what's the complexity of merge sort?

Complexity of Merge Sort

- To determine the time complexity, let’s break our merge sort into pieces and analyze the pieces

- Remember, merge sort consists of:
  - divide a list into two halves
  - sort the halves
  - recombine the sorted halves into a sorted whole

- Dividing the list and recombining the lists are pretty easy to analyze
  - both have O(n) time complexity

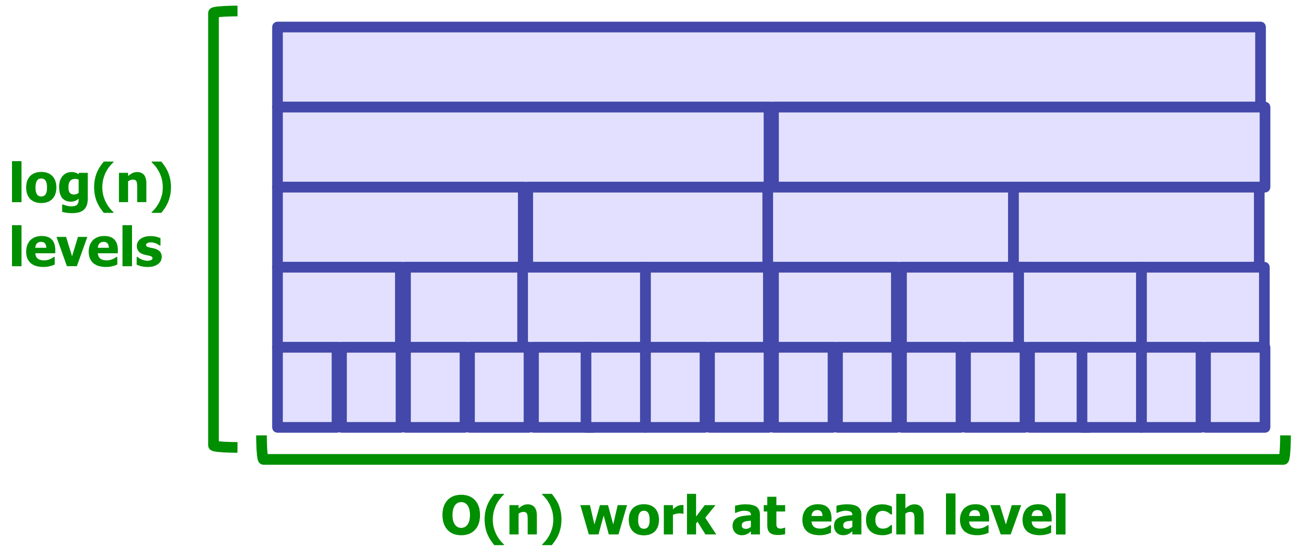
- But what about sorting the halves?

Complexity of Merge Sort

- We can think of merge sort as occurring in levels
  - at the first level, we want to sort the whole list
  - at the second level, we want to sort the two half lists
  - at the third level, we want to sort the four quarter lists
  - ...
- We know there's O(n) work at each level from dividing/ recombining the lists
- But how many levels are there?
  - if we can figure this out, our time complexity is just O(n * num_levels)

Complexity of Merge Sort

- Because we divide the array in half each time, there are log(n) levels



- So merge sort is an O(n log(n)) algorithm
  - this is a big improvement over the $O(n^2)$ sorting algorithms