Revised 12/03/02

Sorting - Merge Sort

Cmput 115 - Lecture 12

Department of Computing Science
University of Alberta

Some code in this lecture is based on code from the book:
Java Structures by Duane A. Bailey or the companion structure package

2/STR/ALG/DEF/2/A

About This Lecture

In this lecture we will learn about a sorting algorithm called the Merge Sort.

We will study its implementation and its time and space complexity.

Outline

Merge: combining two sorted arrays
Merge algorithm
Time and Space complexity for Merge

The Merge Sort Algorithm
Merge Sort - Arrays
Time and Space Complexity of Merge Sort

4/STR/ALG/DEC/1/A

Merging Two Sorted Arrays

Merge is an operation that combines two sorted arrays together into one.

0	1	2
10	40	60

0	1	2	3
50	70	80	90

merge

0	1	2	3	4	5	6
10	40	50	60	70	80	90

Merge Algorithm – initial version

For now, assume the result is to be placed in a separate array called `result`, which has already been allocated.

The two given arrays are called `front` and `back` (the reason for these names will be clear later).

`front` and `back` are in increasing order.

For the complexity analysis, the size of the input, n, is the sum $n_{front} + n_{back}$

Merge Algorithm

For each array keep track of the current position (initially 0).

REPEAT until all the elements of <u>one</u> of the given arrays have been copied into `result` :

- Compare the current elements of `front` and `back`
- Copy the smaller into the current position of `result` (break ties however you like)
- Increment the current position of `result` and the array that was copied from

Copy all the remaining elements of the other given array into `result.`

7/EXP/ALG/VIS/3/A

Merge Example (1)

Current positions indicated in red

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6

Compare current elements; copy smaller; update current

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10						

Compare current elements; copy smaller; update current

Merge Example (2)

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40					

Compare current elements; copy smaller; update current

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40	50				

Compare current elements; copy smaller; update current

Merge Example (3)

All elements copied

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40	50	60			

Copy the rest of the elements from the other array

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40	50	60	70	80	90

10/STR/ALG/DEO/2/A

Merge Code – version 1 (1)

```
private static void merge(int[] front, int[] back,
                int[] result, int first, int last) {
// pre: all positions in front and back are sorted,
//      result is allocated,
//      (last-first+1) == (front.length + back.length)
// post: positions first to last in result contain one copy
// of each element in front and back in sorted order.
 int f=0 ; // front index
 int b=0 ; // back index
 int i=first ; // index in result
 while ( (f < front.length) && (b < back.length)) {
   if (front[f] < back[b]) {
      result[i] = front[f] ;
      i++ ; f++ ;
   } else {
      result[i] = back[b] ;
      i++ ; b++ ;
   }
}
```

Merge Code – version 1 (2)

```
// copy remaining elements into result

while ( f < front.length) {
   result[i] = front[f]
   i++ ;
   f++ ;
 }
while ( b < back.length) {
   result[i] = back[b] ;
   i++ ;
   b++ ;
}
}
```


12/STR/COM/XPL/1/A

Merge – complexity

- Every element in `front` and `back` is copied exactly once. Each copy is two accesses, so the total number of accesses due to copying is 2n.
- The number of comparisons could be as small as min(n_{front},n_{back}) or as large as (n-1). Each comparison is two accesses.
- In the worst case the total number of accesses is 2n+2(n-1) = O(n).
- In the best case the total number of accesses is 2n+ 2*min(n_{front},n_{back}) = O(n)
- The average case is between the worst and best case and is therefore also O(n).
- Memory required: 2n = O(n)

13/STR/ALG/DEF/1/A

Merge Sort Algorithm

Merge Sort sorts a given array (`anArray`) into increasing order as follows:

Split `anArray` into two non-empty parts any way you like. For example

`front` = the first n/2 elements in `anArray`
`back` = the remaining elements in `anArray`

Sort `front` and `back` by recursively calling MergeSort with each one.

Now you have two sorted arrays containing all the elements from the original array. Use `merge` to combine them, put the result in `anArray`.

14/STR/ALG/VIS/5/A

Merge Sort – (1) Split

0	1	2	3	4	5	6
40	60	10	90	50	80	70

0	1	2
40	60	10

0	1	2	3
90	70	80	90

Merge Sort – (2) recursively sort front

0	1	2	3	4	5	6
40	60	10	90	50	80	70

SPLIT

0	1	2
40	60	10

0	1	2	3
90	70	80	90

mergesort(front)

0	1	2
10	40	60

Merge Sort – (3) recursively sort back

0	1	2	3	4	5	6
40	60	10	90	50	80	70

SPLIT

0	1	2
40	60	10

0	1	2	3
90	70	80	90

mergesort(back)

0	1	2
10	40	60

0	1	2	3
50	70	80	90

Merge Sort – (4) merge

Original array

0	1	2	3	4	5	6
40	60	10	90	50	80	70

SPLIT

0	1	2
40	60	10

0	1	2	3
90	70	80	90

Final result

0	1	2	3	4	5	6
10	40	50	60	70	80	90

merge

0	1	2
10	40	60

0	1	2	3
50	70	80	90

Merge Sort Algorithm - summary

Original array

0	1	2	3	4	5	6
40	60	10	90	50	80	70

SPLIT

0	1	2
40	60	10

0	1	2	3
90	70	80	90

Final result

0	1	2	3	4	5	6
10	40	50	60	70	80	90

merge

0	1	2
10	40	60

0	1	2	3
50	70	80	90

Recursively sort each part

19/STR/ALG/DEO/1/A

19

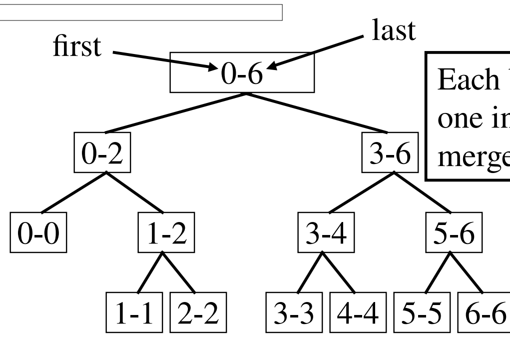
MergeSort Code – version 1

```
public static void mergesort(int[] anArray, int first,
                                            int last) {
//pre:  last < anArray.length
//post: anArray positions first to last are in increasing order
int size = (last-first)+1 ;
if (size > 1) {
  int frontsize = size/2 ;
  int backsize = size-frontsize ;
  int[] front = new int[frontsize] ;
  int[] back  = new int[backsize] ;
  int i;
  for (i=0; i < frontsize; i++) { front[i] = anArray[first+i]; }
  for (i=0; i < backsize;  i++) { back[i]  =
                                  anArray[first+frontsize+i]; }
  mergesort(front,0,frontsize-1);
  mergesort(back, 0,backsize -1);
  merge(front,back,anArray,first,last) ;
  }
}
```


20/EXP/ALG/VIS/2/A

20

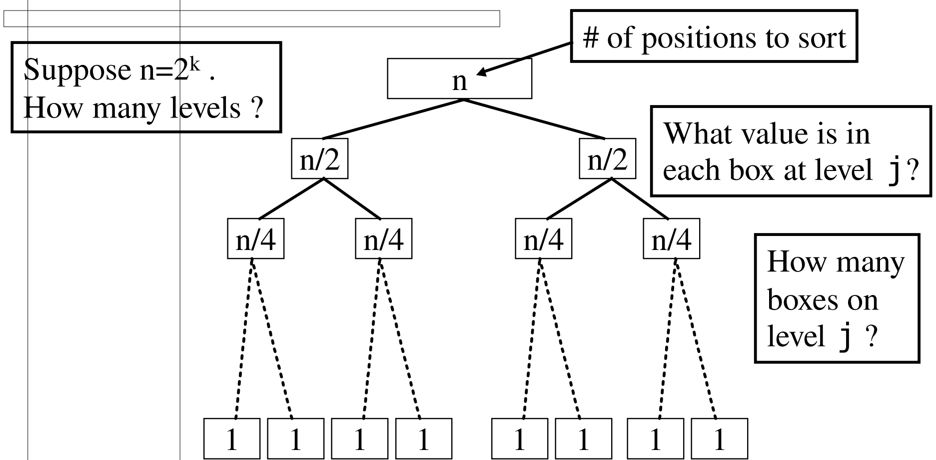
MergeSort Call Graph (n=7)



Each box represents one invocation of the mergesort method.

How many levels are there, in general, if the array is divided in half each time ?

MergeSort Call Graph (general)



22/STR/COM/FAC/1/A

MergeSort – complexity analysis (1)

Each invocation of mergesort on p array positions does the following:

Copies all p positions once (# accesses = O(p))

Calls merge (#accesses = O(p))

Observe that p is the same for all invocations at the same level, therefore total # of accesses at a given level `j` is O((#invocations at level `j`)*p_j)

MergeSort – complexity analysis (2)

The total # of accesses at level `j` is

O((#invocations at level `j`)*p_j)

= O(2^j * (n/2^j))

= O(n)

In other words, the total # of accesses at each level is the same, O(n)

The total # of accesses for the entire mergesort is the sum of the accesses for all the levels. Since the accesses at every level is the same – O(n) – this is

(# levels)*O(n)

= O(log(n))*O(n)

= O(n*log(n))

Time Complexity of Merge Sort

- Best case - O(n log(n))
- Worst case - O(n log(n))
- Average case O(n log(n))
- Note that the insertion sort is actually a better sort than the merge sort if the original collection is almost sorted.

25/STR/COM/XPL/1/A

Space Complexity of Merge Sort (1)

- In any recursive method, space is required for the stack frames created by the recursive calls.
- The maximum amount of memory required for this purpose is
  (size of the stack frame) * (depth of recursion)
- The size of the stack frame is a constant, and for mergesort the depth of recursion (the number of levels) is O(log(n)).
- The memory required for the stack frames is therefore O(log(n)).

26/EXP/COM/XPL/1/A

Space Complexity of Merge Sort (2)

- Besides the given array, there are two temporary arrays allocated in each invocation whose total size is the same as the number of positions to be sorted: at level j this is $p_j = n/2^j$
- This space is allocated <u>before</u> the recursive calls are made and needed <u>after</u> the recursive calls have returned and therefore the maximum total amount of space allocated is the sum of $n/2^j$ for j=0…log(n).
- This sum is O(n) – it is a little less than 2*n.
- Therefore, the space complexity of Merge Sort is O(n), but doubling the collection storage may sometimes be a problem.

27/STR/IMP/DEF/1/A

27

Making mergesort faster

- Although we cannot improve the big-O complexity of mergesort we can make it faster in practice by doing two things:
  – Reducing the amount of copying
  – Allocating temporary storage once at the very outset

- We will make these improvements in 2 steps.

28/EXP/COM/XPL/1/A

28

Reducing copying – `back`

- The `back` array is easy to eliminate. We just use the back portion of `anArray` in its place.
- The only significant change in the code is to the merge method, which now must be told where the “back” of `anArray` begins.
- We can also eliminate from merge the final loop which copies values from `back` into the final positions of `anArray` since these will be in the correct place in `anArray`.

29/STR/ALG/DEO/1/A

29

MergeSort Code – version 2 (1)

```
public static void mergesort(int[] anArray, int first,
                                           int last) {
//pre:  last < anArray.length
//post: anArray positions first to last are in increasing order
int size = (last-first)+1 ;
if ( size > 1) {
  int frontsize = size/2 ;
  int backsize = size-frontsize ;
  int[] front = new int[frontsize] ;
  int[] back  = new int[backsize] ;
  int i;
  for (i=0; i < frontsize; i++) { front[i] = anArray[first+i]; }
  for (i=0; i < backsize;  i++) { back[i]  =
                                  anArray[first+frontsize+i]; }
  mergesort(front,0,frontsize-1);
```


30/EXP/ALG/DEO/3/A

30

MergeSort Code – version 2 (2)

```
mergesort(back, 0,backsize -1);

int backstart = first + frontsize;
mergesort(anArray, backstart, last);

merge(front,back,anArray,first,last) ;

merge(front, anArray, first, backstart, last);

  }
}
```

31

Merge Code – version 2 (1)

```
private static void merge(int[] front,
                int[] anArray, int first, int backstart,
                int last) {
 int f=0 ; // front index
 int b=backstart ; // back index
 int i=first ; // index in result
 while ( (f < front.length) && (b <= last)) {
   if (front[f] < anArray[b]) {
      anArray[i] = front[f] ;
      i++ ; f++ ;
   } else {
      anArray[i] = anArray[b] ;// i <= b ALWAYS AT THIS POINT
      i++ ; b++ ;
   }
}
```


32

Merge Code – version 2 (2)

```
// copy remaining elements into result (anArray)

while ( f < front.length) {
   anArray[i] = front[f]
   i++ ;
   f++ ;
 }
while ( b < back.length) {
   anArray[i] = back[b] ; // i==b ALWAYS AT THIS POINT
   i++ ;
   b++ ;
}
}
```

33/STR/IMP/XPL/1/A

33

Improving efficiency — `front` (1)

- `front` is as easy to eliminate as `back` in the mergesort method. We just use the front portion of `anArray` in its place.
- But the merge method must make a copy of the front portion of `anArray` before merging begins.
- This does not reduce copying at all, but it moves the temporary storage into the merge method, which means it is allocated AFTER the recursive calls and therefore less memory is needed in total.

34/EXP/IMP/XPL/1/A

34

Improving efficiency — `front` (2)

- In addition, instead of allocating the storage each time merge is called, we can allocate it once, before the first call to mergesort is made, and pass this extra array on all calls.
- This saves the time it takes to allocate memory and garbage collect it, which in the previous versions was done once for every invocation.