Lecture 18: Mergesort

Dr Kieran T. Herley

Department of Computer Science
University College Cork

2013/14

Summary

Merge algorithm. List-based MergeSort algorithm. Analysis of same. MergeSort in Java. Array-based Mergesort.

The Merging Problem

Input two *sorted* lists L1 and L2:

$$\begin{array}{lcllll} L1 & : & 1 & 5 & 6 & 7 \\ L2 & : & 2 & 3 & 4 & 8 \end{array}$$

Output Single sorted list containing all values from L1 and L2:

$$L: 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$$

Idea Build up L key by key; at each step remove the smallest remaining key from $L1 \bigcup L2$ and append it to the end of L.

Merge Algorithm

```
Algorithm Merge(L1, L2, L):
   while L1 is not empty and L2 is not empty do /* 1 */
      if L1.get(0) ≤ L2.get(0) then
         L.add(L1.remove(0))
      else
         L.add(L2.remove(0))
   while L1 is not empty do /* 2 */
      L.add(L1.remove(0))
   while L2 is not empty do /* 3 */
      L.add(L2.remove(0))
```

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
    while L1 not empty and L2 not empty do /* 1 */
        if L1.get(0) ≤ L2.get(0) then
            L.add(L1.remove(0))
        else
            L.add(L2.remove(0))
    while L1 is not empty do /* 2 */
        L.add(L1.remove(0))
    while L2 is not empty do /* 3 */
        L.add(L2.remove(0))
```

$L1$: 1 5 6 7

$L2$: 2 3 4 8

L :

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty do /* 1 */
    if L1.get(0) ≤ L2.get(0) then
      L.add(L1.remove(0))
    else
      L.add(L2.remove(0))
  while L1 is not empty do /* 2 */
    L.add(L1.remove(0))
  while L2 is not empty do /* 3 */
    L.add(L2.remove(0))
```

$L1$: 5 6 7

$L2$: 2 3 4 8

L : 1

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty do /* 1 */
    if L1.get(0) ≤ L2.get(0) then
      L.add(L1.remove(0))
    else
      L.add(L2.remove(0))
  while L1 is not empty do /* 2 */
    L.add(L1.remove(0))
  while L2 is not empty do /* 3 */
    L.add(L2.remove(0))
```

$L1$: 5 6 7

$L2$: 3 4 8

L : 1 2

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty do /* 1 */
     if L1.get(0) ≤ L2.get(0) then
        L.add(L1.remove(0))
     else
        L.add(L2.remove(0))
  while L1 is not empty do /* 2 */
     L.add(L1.remove(0))
  while L2 is not empty do /* 3 */
     L.add(L2.remove(0))
```

$L1$: 5 6 7

$L2$: 4 8

L : 1 2 3

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty do /* 1 */
     if L1.get(0) ≤ L2.get(0) then
        L.add(L1.remove(0))
     else
        L.add(L2.remove(0))
  while L1 is not empty do /* 2 */
    L.add(L1.remove(0))
  while L2 is not empty do /* 3 */
    L.add(L2.remove(0))
```

$L1$: 5 6 7

$L2$: 8

L : 1 2 3 4

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty do /* 1 */
    if L1.get(0) ≤ L2.get(0) then
      L.add(L1.remove(0))
    else
      L.add(L2.remove(0))
  while L1 is not empty do /* 2 */
    L.add(L1.remove(0))
  while L2 is not empty do /* 3 */
    L.add(L2.remove(0))
```

$L1$: 6 7

$L2$: 8

L : 1 2 3 4 5

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
   while L1 not empty and L2 not empty do /* 1 */
      if L1.get(0) ≤ L2.get(0) then
         L.add(L1.remove(0))
      else
         L.add(L2.remove(0))
   while L1 is not empty do /* 2 */
      L.add(L1.remove(0))
   while L2 is not empty do /* 3 */
      L.add(L2.remove(0))
```

$L1$: 7

$L2$: 8

L : 1 2 3 4 5 6

Merge Algorithm cont'd

```
Algorithm Merge(L1, L2, L):
    while L1 not empty and L2 not empty do /* 1 */
        if L1.get(0) ≤ L2.get(0) then
            L.add(L1.remove(0))
        else
            L.add(L2.remove(0))
    while L1 is not empty do /* 2 */
        L.add(L1.remove(0))
    while L2 is not empty do /* 3 */
        L.add(L2.remove(0))
```

$L1$:

$L2$: 8

L : 1 2 3 4 5 6 7

Merge Algorithm cont'd

Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty **do** /∗ 1 ∗/
    if L1.get(0) $\leq$ L2.get(0) **then**
      L.add(L1.remove(0))
    else
      L.add(L2.remove(0))
  while L1 is not empty **do** /∗ 2 ∗/
    L.add(L1.remove(0))
  while L2 is not empty **do** /∗ 3 ∗/
    L.add(L2.remove(0))

$L1$:

$L2$:

L : 1 2 3 4 5 6 7 8

Merge cont'd

```
Algorithm Merge(L1, L2, L):
   while L1 not empty and L2
not empty do
      if L1.get(0) ≤ L2.get(0) then
         L.add(L1.remove(0))
      else
         L.add(L2.remove(0))
   while L1 is not empty do /* 2 */
      L.add(L1.remove(0))
   while L2 is not empty do /* 3 */
      L.add(L2.remove(0))
```

Observation

Merge$(L_1, L_2, L3)$ *guarantees that all the elements originally in* L_1 *and* L_2 *are transferred into* L_3 *and that the appear in increasing order within* L_3.

- Merge "empties" contents of $L1$ and $L2$ into L
- Loop 1
  - Transfers values from "front" of $L1$ and $L2$ to "end" of L
  - Smallest remaining value transferred at each step
  - Values added to L in increasing order
- Loops 2 and 3
  - Active only when one of $L1$, $L2$ empty
  - Transfer remaining values from non-empty one to end of L (in increasing order)

Analysis of Merging

Algorithm Merge(L1, L2, L):
  while L1 not empty and L2 not empty **do**
    if L1.get(0) $\leq$ L2.get(0) **then**
      L.add(L1.remove(0))
    else
      L.add(L2.remove(0))
  while L1 is not empty **do** /* 2 */
    L.add(L1.remove(0))
  while L2 is not empty **do** /* 3 */
    L.add(L2.remove(0))

- Algorithm uses List operations isEmpty, get, add, remove
- Note
  - applies get, remove only at start of list
  - applies add only at end of list
- We will ingore any comparisons occuring within the list operations themselves and focus on comparisons embodies within the algorithm *per se*

Analysis of Merging cont'd

Proposition

Merge uses $|L1| + |L2|$ *comparisons*

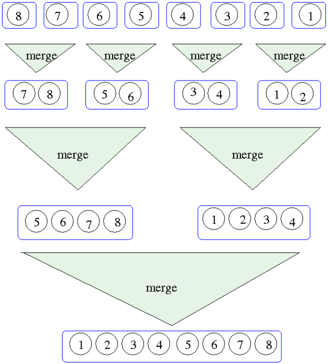
```
Algorithm Merge(L1, L2, L):
   while L1  not empty and L2
not empty do
      if L1.get(0) ≤ L2.get(0) then
         L.add(L1.remove(0))
      else
         L.add(L2.remove(0))
   while L1 is not empty do /* 2 */
      L.add(L1.remove(0))
   while L2 is not empty do /* 3 */
      L.add(L2.remove(0))
```

- Consider List ops. "free" *i.e.* no comparisons
- Loop 1:
  - Each iteration requires one comparison
  - Each iteration removes one element, so at most $|L1| + |L2|$ iterations
  - At most $|L1| + |L2|$ comparisons overall
- Loops 2 and 3– zero comps. each (no comparisons)

Sorting by Merging

Idea

Can sort using carefully chosen pattern of merges!

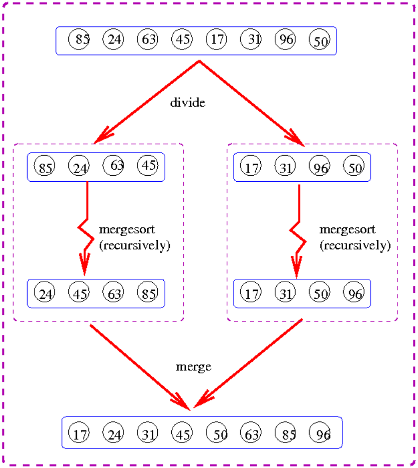


- Mergesort can be formulated as either recursive or non-recursive algorithm
- Recursive version developed here
- See website for non-rec. code

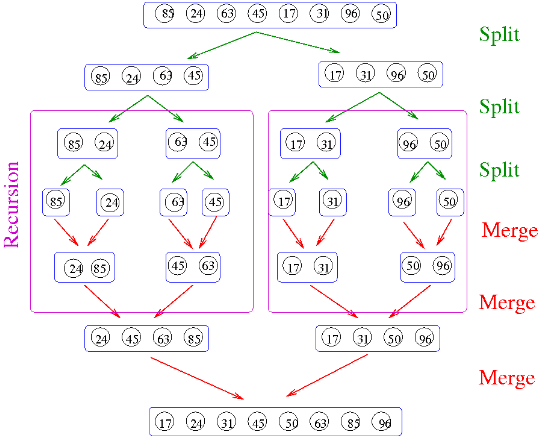
Merge-Sort Idea

To sort list L:

- If L has fewer than two elements, do nothing
- If L has at least two elements,
  - Divide L into two lists L_1 and L_2 of equal size,
  - Recursively sort L_1 and L_2 and
  - Transfer elements back into L by merging (sorted) L_1 and (sorted) L_2



Merge-Sort Idea cont'd



Same computation with all the recursion "unravelled"

Merge-Sort

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

Note: for clarity we assume lists sizes are powers of two. Algorithm works for list of any size, but "splitting loop" needs to be refined to cope with possibility $|L|$ might be odd.

Merge-Sort cont'd

Initial L :
$85, 24, 63, 45, 17, 31, 96, 50$

Algorithm MergeSort (L):
    if |L| > 1 **then**
        h ←|L|/2
        for i ←1 **to** h **do**
            L1.add(L.remove(0))
        for i ←1 **to** h **do**
            L2.add(L.remove(0))
        MergeSort(L1)
        MergeSort(L2)
        Merge(L1, L2, L)

Merge-Sort cont'd

Initial L :

85, 24, 63, 45, 17, 31, 96, 50

After division

L1: 85, 24, 63, 45
L2: 17, 31, 96, 50

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

Merge-Sort cont'd

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

Initial L :

85, 24, 63, 45, 17, 31, 96, 50

After division

L1: 85, 24, 63, 45
L2: 17, 31, 96, 50

After recursive calls

L1: 24, 45, 63, 85
L2: 17, 31, 50, 96

Merge-Sort cont'd

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

Initial L :
$85, 24, 63, 45, 17, 31, 96, 50$

After division

L1: $85, 24, 63, 45$
L2: $17, 31, 96, 50$

After recursive calls

L1: $24, 45, 63, 85$
L2: $17, 31, 50, 96$

After Merge L :
$17, 24, 31, 45, 50, 63, 85, 96$

Proof That Mergesort Works

Assume elements are distinct (argument generalizes)

Proposition

MergeSort(L) sorts elements of L into increasing order.

Observation

Suffices to show for every pair of elements x and y, MergeSort rearranges list so x and y are in correct relative order i.e. smaller before the larger

Observation

Merge($L_1, L_2, L3$) guarantees that all the elements originally in L_1 and L_2 are transferred into L_3 and that they appear in increasing order within L_3.

Proof cont'd

Assume elements are distinct (argument generalizes).

Proposition

MergeSort(L) sorts elements of L into increasing order.

- Proof by induction on $|L|$
- Case $|L| = 1$:

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

Proof cont'd

Assume elements are distinct (argument generalizes).

Proposition

MergeSort(L) sorts elements of L into increasing order.

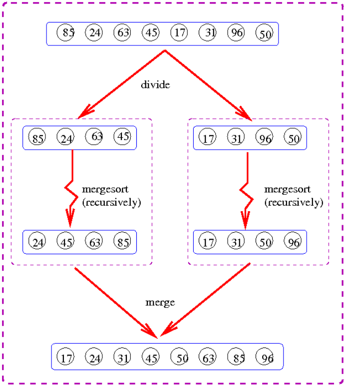
- Proof by induction on $|L|$
- Case $|L| = 1$:

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

- For list size 1 alg. does nothing
- But list is already sorted, so this is correct

Proof cont'd

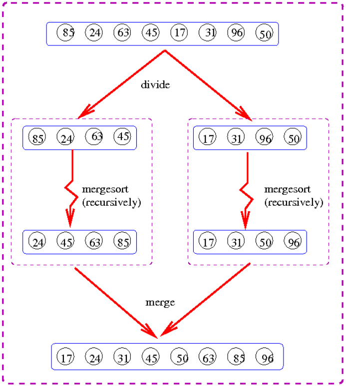
- Case $|L| > 1$:
  - Consider any two elements x and y in S
  - Four possibilities for x, y during division:



- Both go to $L1$

Proof cont'd

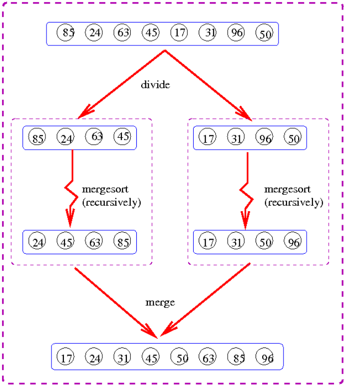
- Case $|L| > 1$:
  - Consider any two elements x and y in S
  - Four possibilities for x, y during division:



- Both go to $L1$– rec. sort on $L1$ places x, y in correct relative order, merge preserves it (implicit induction)
- Both go to $L2$ —akin to above

Proof cont'd

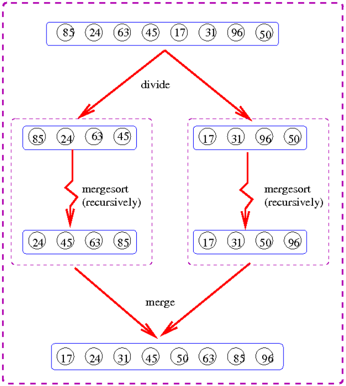
- Case $|L| > 1$:
  - Consider any two elements x and y in S
  - Four possibilities for x, y during division:



- Both go to $L1$– rec. sort on $L1$ places x, y in correct relative order, merge preserves it (implicit induction)
- Both go to $L2$ —akin to above
- x goes to $L1$, y to $L2$

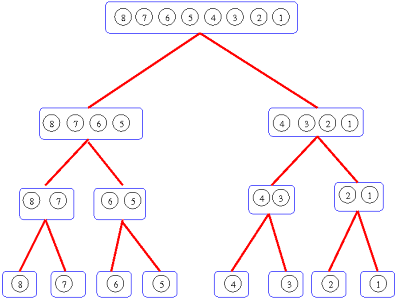
Proof cont'd

- Case $|L| > 1$:
  - Consider any two elements x and y in S
  - Four possibilities for x, y during division:



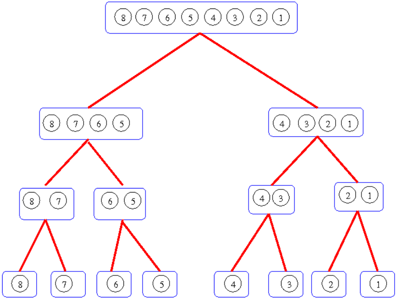
- Both go to $L1$– rec. sort on $L1$ places x, y in correct relative order, merge preserves it (implicit induction)
- Both go to $L2$ —akin to above
- x goes to $L1$, y to $L2$–Merge ensures correct relative order
- y goes to $L1$, x to $L2$ –akin to above

Merge-Sort Recursion Tree



- Top-level involves list of size n giving rise to two recursive calls for lists of size $n/2$,
- Each giving rise to two rec. calls for lists of size $n/4$,
- And so on . . .
- Recursion stops at level i where $n/2^i = 1$, *i.e.* $i = \log n$.

Analysis– Basic Idea



- How to account for all comparisons entailed in algorithm's execution?
- For each MergeSort call
  - Count comparisons associated *directly* with call itself
  - But exclude those associated with recursive calls (to be accounted for separately)
- Add up comparison-counts across all tree nodes.

Analysis of Merge-Sort

```
Algorithm MergeSort (L):
  if |L| > 1 then
    h ←|L|/2
    for i ←1 to h do
      L1.add(L.remove(0))
    for i ←1 to h do
      L2.add(L.remove(0))
    MergeSort(L1)
    MergeSort(L2)
    Merge(L1, L2, L)
```

- (Assume $|L| = n$ is a power of two– argument generalizes)
- $|L| > 1$ one comp.
- Splitting L into $L1$ and $L2$:
  - for loop: $|L|/2$ iterations, $|L|/2 + 1$ comparisons
  - times two *i.e.* $|L| + 2$
- Merge: $|L1| + |L2| = |L|$ comparisons
- (Sub-)total: $2|L| + 3$
- Reflects comps. directly associated with call, but excludes those associate with recursive calls– to be accounted for separately

Analysis of Merge-Sort cont'd

Proposition

MergeSort uses $\approx 2n\log n$ *comparisons*

- Level i of rec. tree has 2^i MergeSort calls/invocations, each for list of size $n/2^i$.

Analysis of Merge-Sort cont'd

Proposition

MergeSort uses $\approx 2n \log n$ *comparisons*

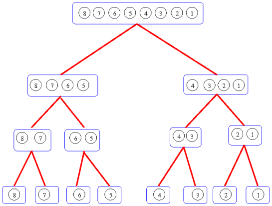
- Level i of rec. tree has 2^i MergeSort calls/invocations, each for list of size $n/2^i$.
- Each level-i invocation uses $2 \cdot n/2^i + 3$ comps. for: (i) dividing list into two halves; (ii) merging two (sorted) halves.

Analysis of Merge-Sort cont'd

Proposition

MergeSort uses $\approx 2n \log n$ *comparisons*

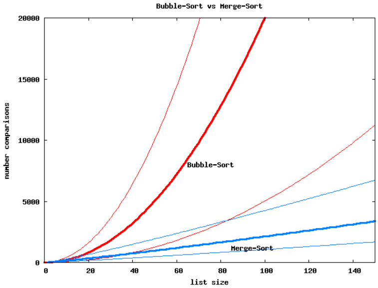
- Level i of rec. tree has 2^i MergeSort calls/invocations, each for list of size $n/2^i$.
- Each level-i invocation uses $2 \cdot n/2^i + 3$ comps. for: (i) dividing list into two halves; (ii) merging two (sorted) halves.



Summing over all nodes at various levels

$$\begin{aligned}\sum_{i=0}^{\log n} (2 \cdot n/2^i + 3) \cdot 2^i &= \sum_{i=0}^{\log n} 2n + \sum_{i=0}^{\log n} 3 \cdot 2^i \\ &= 2n(1 + \log n) + 3(2^{1+\log n} - 1) \\ &= 2n \log n + 8n - 3\end{aligned}$$

Merge-Sort vs Bubble-Sort



- Number of comparisons:

  Merge-Sort
  $2n \log n + 8n - 3$
  Bubble-Sort $2n^2 + 1$

- n^2 grows far more rapidly than $n \log n$
- Suggests MS dramatically more efficient than BS for large n

Merge-Sort in Java

```
public interface SortObject<EltType>
{   public void sort ( List <EltType> L, Comparator<EltType> c);
}
```

```
public class MergeSort<EltType>
        implements SortObject<EltType>
{   public void sort ( List <EltType>, Comparator<EltType> c)
    {    /* Java implementation of MergeSort */ }
    . . .
}
```

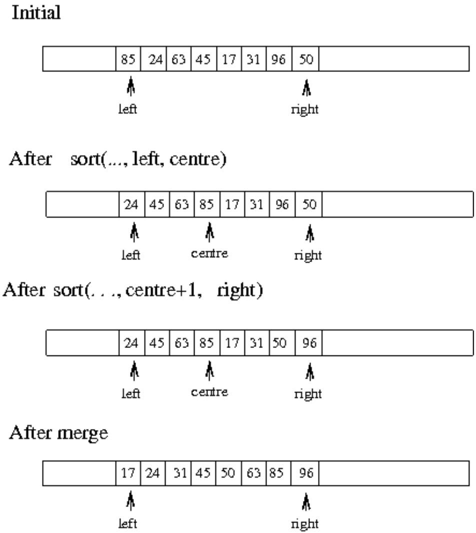
Same interface as Bubblesort, internals different

Array-Based Mergesort

- ```
  public interface SortObject2<EltType>
  {  public void sort(EltType[] L, Comparator<EltType> c); }
  ```

- ```
  public class MergeSort2<EltType>
         implements SortObject2<EltType>
  {  public void sort(EltType[] a, Comparator<EltType> c)
     { . . .}
     . . .
  }
  ```

Array-Based Mergesort cont'd



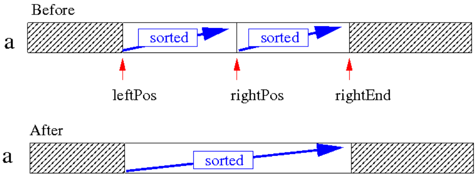
- Use input array and segments thereof in lieu of lists (A[left $\cdots$ right], A[left $\cdots$ centre, A[certre+1 $\cdots$ right' represent L, $L1$, $L2$ respectively)
- Use one temp array to simplify merge code

Array-Based Mergesort cont'd

```
public void sort(EltType[] a, Comparator<EltType> c)
{  EltType[] tmp = (EltType[]) new Object[a.length];
   sort(a, tmp, c, 0, a.length-1);
}
private void sort(EltType[] a, EltType[] tmp,
      Comparator<EltType> c, int left, int right)
{  if (left < right)
   {  int centre = (left + right)/2;
      sort(a, tmp, c, left, centre);
      sort(a, tmp, c, centre+1, right);
      merge(a, tmp, c, left, centre+1, right);
   }
}
```

Array-Based Mergesort cont'd

```
public void merge(
    EltType[] a, EltType[] tmp, Comparator<EltType> c,
    int leftPos, int rightPos, int rightEnd)
{ . . . }
```



Builds up merged list in tmp, then copies it back into a