Computer Science 331
Merge Sort

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #23

Outline

Introduction

Merge Sort is is an asymptotically faster algorithm than the sorting algorithms we have seen so far.

- It can be used to sort an array of size n using $\Theta(n \log_2 n)$ operations in the worst case.

Presented here: A version that takes an input array A and produces another sorted array B (containing the entries of A, rearranged)

A solution to the "Merging Problem" (presented next) is a subroutine that is used to do much of the work.

Reference: Textbook, Section 10.7

The "Merging" Problem

Calling Sequence: merge(A_1, A_2, B)

Precondition:

- A_1 is a sorted array of length $n_1 \in \mathbb{N}$, so that

$$A_1[h] \leq A_1[h+1] \qquad \text{for } 0 \leq h \leq n_1 - 2$$

- A_2 is a sorted array of length $n_2 \in \mathbb{N}$, so that

$$A_2[h] \leq A_2[h+1] \qquad \text{for } 0 \leq h \leq n_2 - 2$$

- Entries of A_1 and A_2 are objects of the same ordered type

The "Merging" Problem (cont.)

Postcondition:

- B is a sorted array of length $n_1 + n_2$, so that

$$B[h] \le B[h+1] \qquad \text{for } 0 \le h \le n_1 + n_2 - 2$$

- Entries of B are the entries of A_1 together with the entries of A_2, reordered but otherwise unchanged
- A_1 and A_2 have not been modified

Idea for an Algorithm

Maintain indices into each array (each initially pointing to the leftmost element)

repeat

- Compare the current elements of each array
- Append the smaller entry onto the "end" of B, advancing the index for the array from which this entry was taken

until one of the input arrays has been exhausted

Append the rest of the other input array onto the end of B

Pseudocode

merge(A_1, A_2, B)
  $n_1 = length(A_1)$; $n_2 = length(A_2)$
  Declare B to be an array of length $n_1 + n_2$
  $i_1 = 0$; $i_2 = 0$; $j = 0$
  while $(i_1 < n_1)$ **and** $(i_2 < n_2)$ **do**
    if $A_1[i_1] \le A_2[i_2]$ **then**
      $B[j] = A_1[i_1]$; $i_1 = i_1 + 1$
    else
      $B[j] = A_2[i_2]$; $i_2 = i_2 + 1$
    end if
    $j = j + 1$
  end while

Pseudocode, Continued

$\{$Copy remainder of A_1 (if any)$\}$
while $i_1 < n_1$ **do**
  $B[j] = A_1[i_1]$; $i_1 = i_1 + 1$; $j = j + 1$
end while

$\{$Otherwise copy remainder of $A_2\}$
while $i_2 < n_2$ **do**
  $B[j] = A_2[i_2]$; $i_2 = i_2 + 1$; $j = j + 1$
end while

Example

A_1 : | 1 | 3 | 5 | 6 | 8 | A_2 : | 4 | 5 | 7 | 9 | 10 |

$j = 0, i_1 = 0, i_2 = 0$

-

B: | | | | | | | | | | |

$j = 1, i_1 = \quad, i_2 =$

-

B: | | | | | | | | | | |

$j = 2, i_1 = \quad, i_2 =$

-

B: | | | | | | | | | | |

Example (cont.)

$j = 3, i_1 = \quad, i_2 =$ B: | | | | | | | | | | |

$j = 4, i_1 = \quad, i_2 =$ B: | | | | | | | | | | |

$j = 5, i_1 = \quad, i_2 =$ B: | | | | | | | | | | |

$j = 6, i_1 = \quad, i_2 =$ B: | | | | | | | | | | |

$j = 7, i_1 = \quad, i_2 =$ B: | | | | | | | | | | |

$j = 8, i_1 = \quad, i_2 =$

Final result: B: | | | | | | | | | | |

Loop Invariant for Loop #1

After the k^{th} execution of the body of the first loop

- $n_1 = \mathit{length}(A_1) \in \mathbb{N}$; $n_2 = \mathit{length}(A_2) \in \mathbb{N}$; $i_1, i_2, j \in \mathbb{N}$;
- $0 \leq i_1 \leq n_1$ and $0 \leq i_2 \leq n_2$;
- $j = k = i_1 + i_2$;
- $B[h] \leq B[h+1]$ for $0 \leq h \leq j-2$;
- $B[0], B[1], \ldots, B[j-1]$ are the values

$$A_1[0], A_1[1], \ldots, A_1[i_1 - 1] \quad \text{and} \quad A_2[0], A_2[1], \ldots, A_2[i_2 - 1],$$

  reordered but otherwise unchanged;
- if $j \geq 1$ and $i_1 < n_1$ then $B[j-1] \leq A_1[i_1]$
- if $j \geq 1$ and $i_2 < n_2$ then $B[j-1] \leq A_2[i_2]$
- The arrays A_1 and A_2 have not been changed.

Analysis for Loop #1, Concluded

Loop Variant for Loop #1: $f(n_1, n_2, j) = n_1 + n_2 - j$

- loop invariant implies that $j = i_1 + i_2$, so that

$$f(n_1, n_2, j) = (n_1 - i_1) + (n_2 - i_2)$$

- it follows, since $i_1 \leq n_1$ and $i_2 \leq n_2$, that $i_1 = n_1$ and $i_2 = n_2$ if $f(n_1, n_2, j) \leq 0$
- Initial value of $j = 0$, so worst-case # of iterations is $\leq n_1 + n_2$

Application of Loop Invariant and Loop Variant: failure of the loop test, and the loop invariant, implies that either $i_1 = n_1$ or $i_2 = n_2$ on termination of the first loop. Conclusion:

-
-
-

Analysis for Loop #2

Loop Invariant for Loop #2:
Same as the loop invariant for loop #1, along with the additional condition

$$\text{either } i_1 = n_1 \text{ or } i_2 = n_2$$

Loop Variant for Loop #2: Same as for loop #1.

Conclusions:

-
-

Analysis for Loop #3

Loop Invariant for Loop #3:
Same as the loop invariant for loop #1, along with the additional condition

$$i_1 = n_1$$

Loop Variant for Loop #3: Same as for loop #1.

Conclusions:

-
-

Analysis of the Merging Algorithm

Correctness:

- loop invariants prove partial correctness (if **merge** terminates, the output is correct)
- loop variant implies that the for loops (and hence the entire algorithm) terminate
- therefore, **merge** is correct

Efficiency:

- Each of the three loops executes no more than $n_1 + n_2$ times
- each loop body requires a constant number of steps
- total cost of **merge** is $\Theta(n_1 + n_2)$

Merge Sort: Idea for an Algorithm

Suppose we:

1. Split an input array into two roughly equally-sized pieces.
2. *Recursively* sort each piece.
3. Merge the two sorted pieces.

This sorts the originally given array.

Note: this algorithm design strategy is known as *divide-and-conquer*:

- divide the original problem (sorting an array) into smaller subproblems (sorting smaller arrays)
- solve the smaller subproblems *recursively*
- combine the solutions to the smaller subproblems (the sorted subarrays) to obtain a solution to the original problem (merging the sorted arrays)

Pseudocode

Merge Sort(A, B)
$n = length(A)$ {Assumption: $n \geq 1$}
if $n == 1$ **then**
$B[0] = A[0]$ {B created with length 1}
else
$n_1 = \lceil n/2 \rceil$
$n_2 = n - n_1$ {so that $n_2 = \lfloor n/2 \rfloor$}
Set A_1 to be $A[0], \ldots, A[n_1 - 1]$ (length n_1)
Set A_2 to be $A[n_1], \ldots, A[n-1]$ (length n_2)
mergeSort(A_1, B_1)
mergeSort(A_2, B_2)
merge(B_1, B_2, B)
end if

Example

	0	1	2	3	4	5	6	7
A :	7	3	9	6	5	2	1	8

1. Sort $A[0, \ldots, 3] = [7, 3, 9, 6]$ recursively:
   -
     -
     -
     -
   -
   -
2. Sort $A[4, \ldots, 7] = [5, 2, 1, 8]$ recursively.
3. Merge: result is $[1, 2, 3, 5, 6, 7, 8, 9]$

Partial Correctness

Theorem 1

*If **mergeSort** is run on an input array A of size n, then either*

- *the algorithm eventually halts, producing the desired sorted array B as output,*

or

- *the algorithm does not halt at all.*

Prove by induction on n

- prove partial correctness for $n = 1$
- prove partial correctness for $n > 1$ assuming partial correctness for arrays of size k for $k < n$

Proof of Partial Correctness

Base Case: $n = 1$

- if $n = 1$, array consists of one element (array is sorted trivially)
- algorithm returns B containing a copy of the single element in the array (terminates with correct output)

Inductive hypothesis:

- assume the algorithm is partially correct for arrays of size $k < n$

Prove that B is sorted under this assumption:

-
-
-
-

Termination and Efficiency

Let $T(n)$ be the number of steps used by this algorithm when given an input array of length n, in the worst case.

We can see the following by inspection of the code:

$$T(n) \leq \begin{cases} c_0 & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c_1 n & \text{if } n \geq 2 \end{cases}$$

for some constants c_0 and c_1.

Special Case: If $n = 2^k$ is a power of two, we can rewrite this as

$$T(n) \leq \begin{cases} c_0 & \text{if } n = 1 \\ 2T(n/2) + c_1 n & \text{if } n \geq 2 \end{cases}$$

Termination and Efficiency

Theorem 2

If $n = 2^k$ is a power of two, and $c = \max(c_0, c_1)$, then

$$T(n) \leq cn \log_2(2n) = cn(k+1).$$

Prove by induction on k

- Base case ($k = 0$): for $k = 0$ we have $n = 2^0 = 1$, and

$$T(1) = c_0 \leq cn(k+1) = c$$

  because $c = \max(c_0, c_1)$

Termination and Efficiency

Inductive hypothesis: Assume $k > 0$ and theorem holds for $k - 1$:

Show that the theorem holds for k :

Termination and Efficiency (General Case)

Consider the function $L(n) = \lceil \log_2 n \rceil$ for $n \geq 1$

Useful Property:

- $L(\lceil n/2 \rceil) = L(n) - 1$ and $L(\lfloor n/2 \rfloor) \leq L(n) - 1$ for every integer $n \geq 2$

Theorem 3

If $n \geq 1$ then $T(n) \leq cnL(2n) \leq cn(\log_2 n + 2)$.

Method of Proof: induction on n

Further Observations

It can be shown (by consideration of particular inputs) that the worst-case running time of this algorithm is also in $\Omega(n \log_2 n)$. It is therefore in $\Theta(n \log_2 n)$

- This is preferable to the classical sorting algorithms, for sufficiently large inputs, if worst-case running time is critical
- The classical algorithms are *faster* on sufficiently *small* inputs because they are simpler

Alternative Approach: A “hybrid” algorithm:

- Use the recursive strategy given above when the input size is greater than or equal to some (carefully chosen) “threshold” value
- Switch to a simpler, nonrecursive algorithm (that is faster on small inputs) as soon as the input size drops to below this “threshold” value