Computer Science 331
Merge Sort

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #24

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.

The "Merging" Problem

Calling Sequence: `void merge(int [] A₁, int [] A₂, int [] B)`

Precondition:

- A_1 is a sorted array of length n_1 (positive integer) such 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 (positive integer) such 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 integers (more generally, objects from the same ordered class)

The "Merging" Problem (cont.)

Postcondition:

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

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

- 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

```
void merge(int [] A₁, int [] A₂, int [] B)
    n₁ = length(A₁); n₂ = length(A₂)
    Declare B to be an array of length n₁ + n₂
    i₁ = 0; i₂ = 0; j = 0
    while (i₁ < n₁) and (i₂ < n₂) do
        if A₁[i₁] ≤ A₂[i₂] then
            B[j] = A₁[i₁]; i₁ = i₁ + 1
        else
            B[j] = A₂[i₂]; i₂ = i₂ + 1
        end if
        j = j + 1
    end while
```
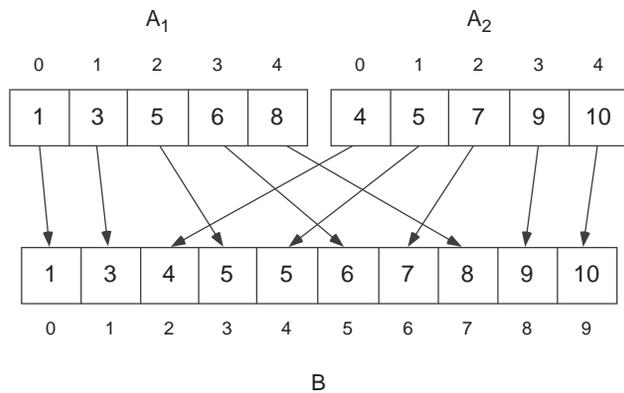
Pseudocode, Continued

```
{Copy remainder of A₁ (if any)}
while i₁ < n₁ do
    B[j] = A₁[i₁]; i₁ = i₁ + 1; j = j + 1
end while

{Otherwise copy remainder of A₂}
while i₂ < n₂ do
    B[j] = A₂[i₂]; i₂ = i₂ + 1; j = j + 1
end while
```

Example

Note: Running time is $\Theta(n_1 + n_2)$, where the input arrays have size n_1 and 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

```
void mergeSort(int [] A, int [] B)
    n = A.length
    if n == 1 then
        B[0] = A[0]
    else
        n₁ = ⌈n/2⌉
        n₂ = n − n₁ {so that n₂ = ⌊n/2⌋}
        Set A₁ to be A[0], ..., A[n₁ − 1] {length n₁}
        Set A₂ to be A[n₁], ..., A[n − 1] {length n₂}
        mergeSort(A₁, B₁)
        mergeSort(A₂, B₂)
        merge(B₁, B₂, B)
    end if
```

Example

$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
A: & 7 & 3 & 9 & 6 & 5 & 2 & 1 & 8
\end{array}
$$

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

Correctness of MergeSort

> **Theorem 1**
>
> *If* **mergeSort** *is run on an input array A of size $n \geq 1$, then the algorithm eventually halts, producing the desired sorted array as output.*

Prove by (strong) induction on n (assuming that **merge** is correct!):

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)

Correctness, continued

Inductive hypothesis:

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

Let A be an array of length $n \geq 2$. Prove that B is sorted copy of A.

- A_1 contains first n_1 elements of A sorted
- A_2 contains remaining n_2 elements of A
- $n_1 = \lceil n/2 \rceil < n$ and $n_2 = \lfloor n/2 \rfloor < n$, so inductive hypothesis implies that B_1 is A_1 sorted and B_2 is A_2 sorted
- **merge** computes B containing all elements of A sorted (assuming that **merge** is correct)
- hence, algorithm is partially correct by induction.   □

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 $c_0 > 0$ and $c_1 > 0$.

Useful Relation:

- For any integer n: $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$.

Termination and Efficiency, continued

Recall That:

- Operators $\lceil \rceil$ and $\lfloor \rfloor$ do not change the asymptotic behavior.

$T(n)$ can be rewritten as follows:

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

Termination and Efficiency, continued

Recurrence Substitution for $n \geq 2$:

$$\begin{aligned}
T(n) &\leq 2T(n/2) + c_1 n \\
&\leq 2\left(2T(n/2^2) + c_1 n/2\right) + c_1 n \\
&= 2^2 T(n/2^2) + 2c_1 n \\
&\leq \cdots \\
&\leq 2^k T\left(\frac{n}{2^k}\right) + kc_1 n
\end{aligned}$$

Termination: $\frac{n}{2^k} = 1 \implies k = \log_2 n$

Termination and Efficiency, concluded

Theorem 2

$$T(n) \leq nc_0 + (n \log_2 n)c_1.$$

Prove by induction on n
- Base case ($n = 1$)
- Inductive Step ($n \geq 2$)

Conclusion: Worst-case is in: $O(n \log_2 n)$

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.

References

Further Reading:

Introduction to Algorithms, Chapter 2.3