Merging, Merge Sort, and Divide-and-Conquer

David Fernández-Baca

Computer Science 311
Iowa State University

February 5, 2022

Merging two Arrays

Given: Two sorted arrays B and C.

Return: A sorted array containing the elements of B and C.

```
Merge(B, C)
    p = B.length, q = C.length
    create an empty array D of length p + q
    i = 0, j = 0
    while i < p and j < q do
        if B[i] ≤ C[j] then
            append B[i] to D
            i++
        else
            append C[j] to D
            j++
    if i ≥ p then
        for k = j to q − 1 do
            append C[k] to D
    else
        for k = i to p − 1 do
            append B[k] to D
    return D
```

Correctness of `Merge`(B, C)

Loop Invariant

At the beginning of each iteration of the **while** loop, $\langle D[0], \ldots, D[i+j-1] \rangle$ consists of the elements of $\langle B[0], \ldots, B[i-1] \rangle$ and $\langle C[0], \ldots, C[j-1] \rangle$ in sorted order.

Correctness of $\texttt{Merge}(B, C)$

Two possibilities at termination of the **for** loop:

- $i = p$: Loop invariant implies that all elements of $\langle B[0] \ldots, B[p-1]\rangle = B$ and $\langle C[0], \ldots, C[j-1]\rangle$ are in sorted order D, but $\langle C[j], \ldots, C[q-1]\rangle$ remain to be inserted into D.
- $j = q$: Loop invariant implies that all elements $\langle C[0] \ldots, C[q-1]\rangle = C$ and $\langle B[0], \ldots, B[j]\rangle$ are in sorted order in D, but $\langle B[j], \ldots, B[p-1]\rangle$ remain to be inserted into D.

The final **if-then-else** statement adds the un-inserted elements to D.

Running Time of `Merge`

- Let $n = p + q$, where $p = B.\texttt{length}$ and $q = C.\texttt{length}$.
- Since each iteration of the while takes constant time, the loop takes $O(n)$ time.
- Appending the remainder of B or C to D also takes $O(n)$ time.

$\Rightarrow$ `Merge` takes $O(n)$ time.

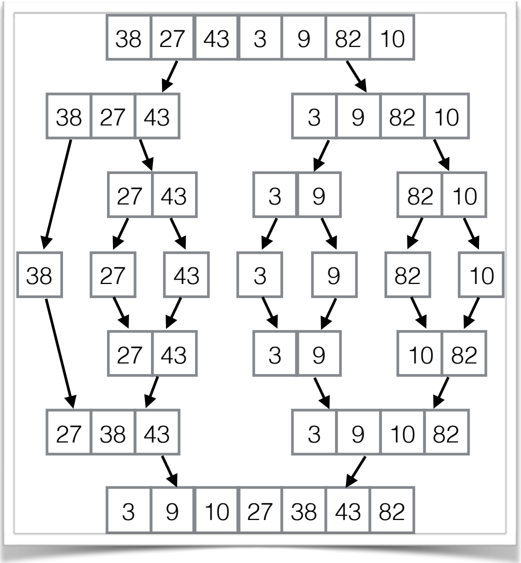
Note

`Merge` requires $\Theta(n)$ additional space for temporary storage.

Merge Sort

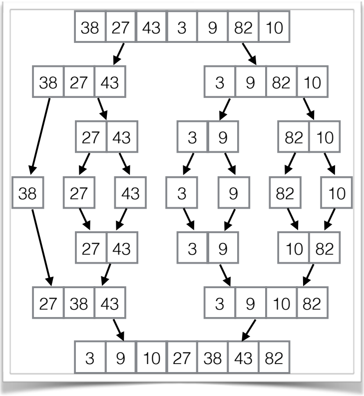
```
MergeSort(A)
    n = A.length
    if n == 1 then
        return A
    A_left = ⟨A[0], ..., A[⌊n/2⌋ − 1]⟩
    A_right = ⟨A[⌊n/2⌋], ..., A[n − 1]⟩
    A = Merge(MergeSort(A_left), MergeSort(A_right))
    return A
```

Example


38 27 43 3 9 82 10
38 27 43
3 9 82 10
27 43
3 9
82 10
38
27
43
3
9
82
10
27 43
3 9
10 82
27 38 43
3 9 10 82
3 9 10 27 38 43 82

Running Time of `MergeSort`

- Each level of the recursion tree involves $O(n)$ operations.
- There are $O(\log n)$ levels.
- Hence, `MergeSort` runs in $O(n \log n)$ time.



Divide-and-Conquer

Divide-and-Conquer

Components of a Divide-and-Conquer Algorithm

Divide: Break the problem into a number of subproblems that are smaller instances of the same problem.

Conquer: Solve subproblems recursively.

- If a subproblem is sufficiently small, solve it directly, without recursion, in $\Theta(1)$ time.

Combine: Merge the solutions to the subproblems into the solution for the original problem.

Divide-and-Conquer Recurrences

- A recurrence equation describes the running time of a recursive algorithm on problem of size n in terms of its running time on smaller inputs.
- Let $T(n)$ be the running time on a problem of size n.
- If the problem is small enough, say $n \le c$ for some c, the algorithm takes $\Theta(1)$ time.
- Suppose larger problems are divided into a subproblems, each of size n/b.
- Let $D(n)$ be the time to divide the problem into subproblems and $C(n)$ be the time to combine their solutions.
- Then,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

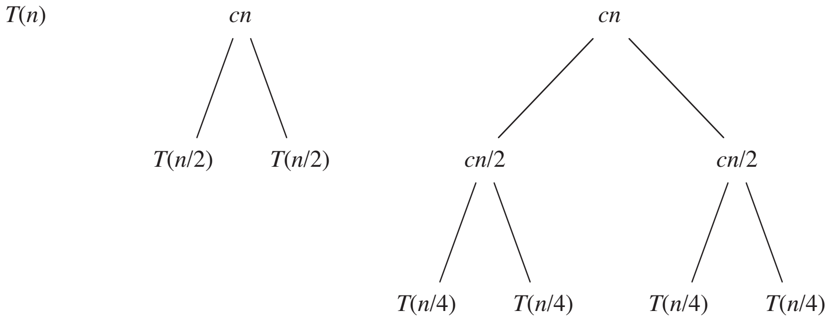
A Recurrence for `MergeSort`

- `MergeSort` splits the input into $a = 2$ subproblems.
- Each subproblem has size $n/2$, so $b = 2$.
- Splitting array in 2 and merging results after recursive calls takes cn time, for some constant c; i.e., $C(n) + D(n) = cn$.
- Thus,

$$T(n) = \begin{cases} c & \text{if } n \leq 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

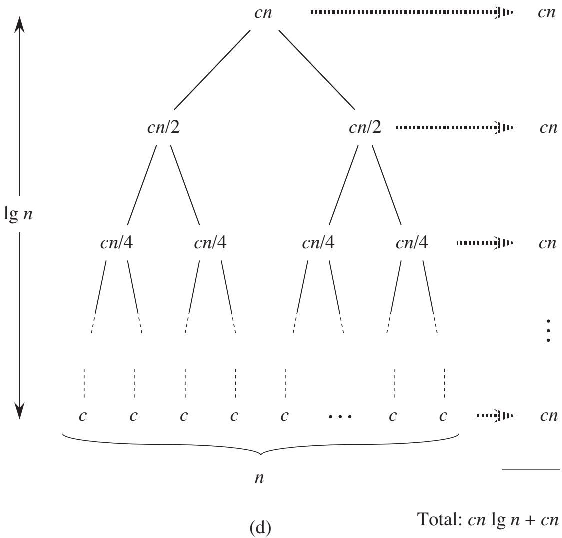
- We can prove that $T(n) = \Theta(n \log n)$ by expanding the recurrence.

Expanding the Recurrence



Source: CLRS

Expanding the Recurrence



(d)

Source: CLRS

Floors and Ceilings

Our analysis of `MergeSort` assumed that n is always divisible by 2, so

$$T(n) = \begin{cases} c & \text{if } n \leq 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases} \tag{1}$$

But (1) is a simplification. A more accurate recurrence is

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n > 1. \end{cases} \tag{2}$$

It turns out that we lose nothing from the simplification.

Fact

If $T(n)$ is defined by (2), then $T(n) = \Theta(n \log n)$.

See CLRS, Section 4.6.2.

Bibliography

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (3rd edition), MIT Press, 2009.

[KT] Jon Kleinberg and Éva Tardos, *Algorithm Design*, Addison-Wesley, 2006.