Lecture 8: Sorting I: Insertion Sort, Merge Sort, Master Theorem

Lecture Overview

- Sorting
- Insertion Sort
- Mergesort (Divide and Conquer)
- In-Place Sorting
- Master Theorem

Readings

CLRS Chapter 4

The Sorting Problem

Input: An array $A[0:n]$ containing n numbers in $\mathbb{R}$.
Output: A sorted array $B[0:n]$ containing the same numbers.

e.g. $A = [7, 2, 5, 5, 9.6] \rightarrow B = [2, 5, 5, 7, 9.6]$

many applications, e.g.: phonebook.

Sorting Methods

Insertion Sort

for $i = 1, 2, \ldots, n$

- insert element $A[i]$ into the sorted array $A[0:i]$ by pairwise swaps down to its right position.

E.g. Sample Execution: See Figure 1.

Running Time?
$O(n^2)$, **worst case example:** $A = [n, n-1, n-2, \ldots, 2, 1]$.

Improve to $O(n \log n)$?
Replace downward pairwise swaps, with binary search in $A[0:i]$.
Called Binary Insertion Sort.

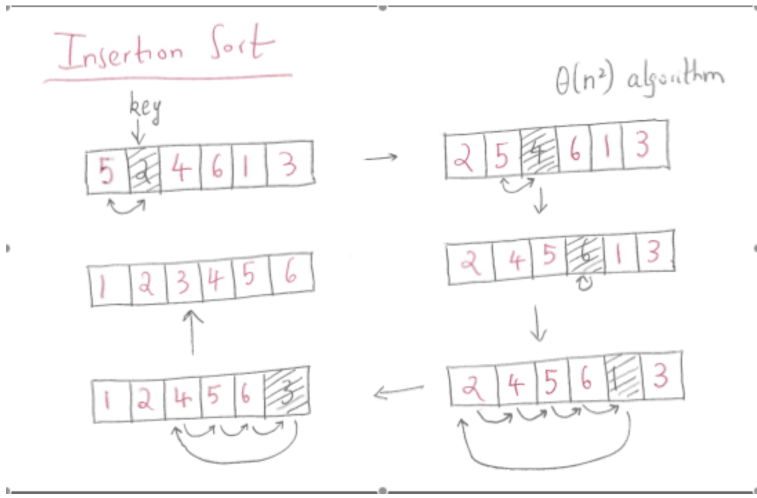

Figure 1: Insertion Sort Sample Execution

Merge Sort

Sorting Algorithm that uses the Divide & Conquer paradigm. See Figure 2

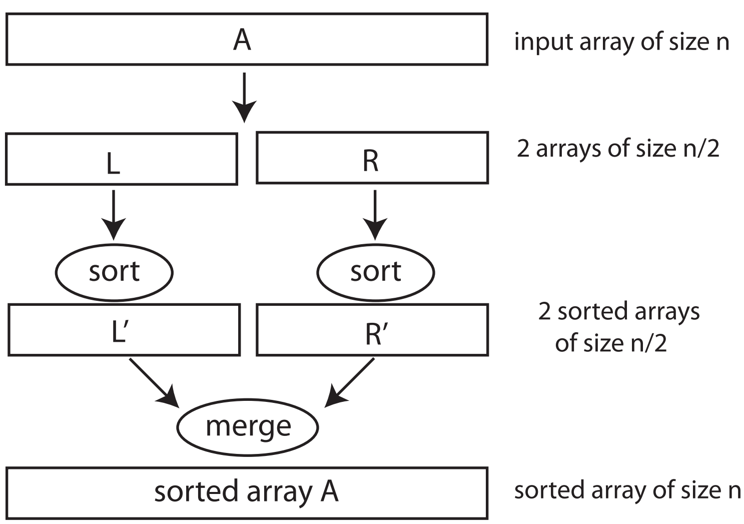

Figure 2: Divide/Conquer/Combine Paradigm

Fast Merge: Exploit the fact that the arrays are already sorted. “Two finger” algorithm—Figure 3—takes linear time.

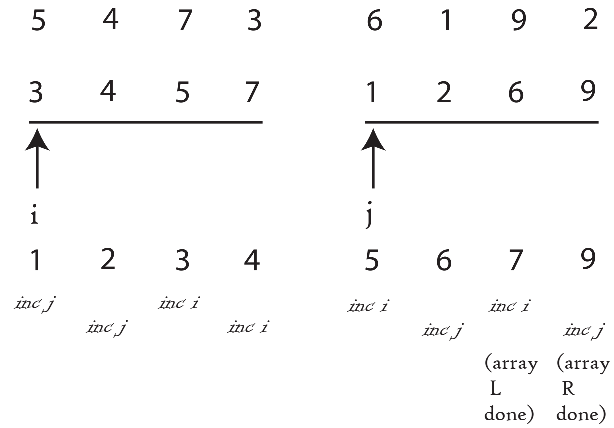

Figure 3: "Two Finger" Algorithm for Merge

Key Property: *Sort* is done recursively.

Run-time Analysis

$$T(n) = \underbrace{C_1}_{divide} + \underbrace{2.T(n/2)}_{recursion} + \underbrace{C.n}_{merge}$$

Unravelling the recursion

$$\begin{aligned}
T(n) &= 2T(n/2) + C \cdot n + C_1 \\
&= 2\left(2T(n/4) + C \cdot \frac{n}{2} + C_1\right) + C \cdot n + C_1 = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2C \cdot n + (1+2)C_1 \\
&= \ldots = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3C \cdot n + (1+2+2^2)C_1 \\
&= \ldots \\
&= 2^k \cdot T\left(\frac{n}{2^k}\right) + kC \cdot n + (1+2+2^2+\ldots+2^{k-1})C_1 = \\
&= (\text{assuming } n = 2^k) = nT(1) + \log_2 n \cdot C \cdot n + (n-1)C_1 = \Theta(n \log_2 n).
\end{aligned}$$

See Figure 4: The leaves correspond to matrices of size 1 at the maximum recursion depth (no further division into subproblems is possible). Going bottom-up in the recursion tree, need to pay the merge cost and the divide cost. The depth of the tree is $\Theta(\log n)$ and every level costs $\Theta(n)$. Total is $\Theta(n \log n)$. We omitted the constant additive cost C_1 from the nodes in the figure.

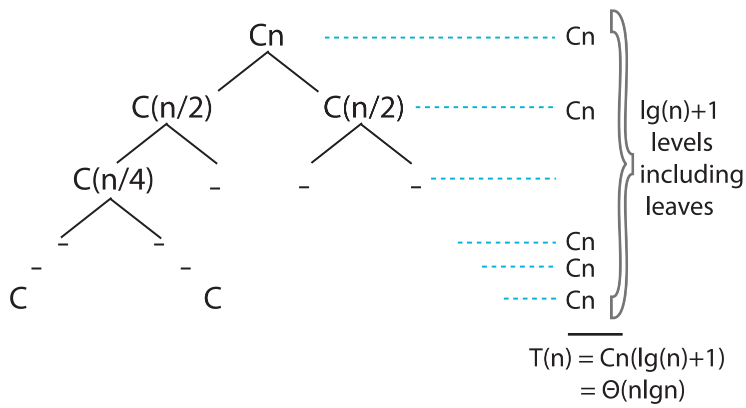

Figure 4: Recursive Structure of Merge Sort

An Experiment

Insertion Sort	$\Theta(n^2)$
Merge Sort	$\Theta(n \log(n))$

- Test Merge Routine: Merge Sort (in Python) takes $\approx 2.2n\log(n)$ μs
- Test Insert Routine: Insertion Sort (in Python) takes $\approx 0.2n^2$ μs
- Test Insert Routine: Insertion Sort (in C) takes $\approx 0.01n^2$ μs

Question: When is Merge Sort (in Python) $2n\lg(n)$ better than Insertion Sort (in C) $0.01n^2$?
Answer: Merge Sort wins for $n \geq 2^{12} = 4096$
Take Home Point: A better algorithm is sometimes more valuable than hardware or compiler improvements even for modest n.

In-Place Sorting

Numbers re-arranged in the input array A with at most a constant amount of extra storage at any time.

Insertion Sort: only $O(1)$ extra space is needed; so in-place

Merge Sort: need $O(n)$ auxiliary space during merging and (depending on the underlying architecture) may require up to $\Theta(n\log n)$ space for the stack. Can turn it into an in-place sorting algorithm by designing the algorithm more carefully.

Master Theorem

Generic Divide and Conquer Recursion:

$$T(n) = aT(n/b) + f(n),$$

where

- a is the number of subproblems
- n/b is the size of each subproblem—hopefully $b > 1$
- $f(n)$ is the cost of dividing the problem into subproblems, and merging the solutions of the subproblems.

E.g. 1 Mergesort: $a = 2$, $b = 2$, $f(n) = Cn + C_1$.
E.g. 2 Binary Search: $a = 1$, $b = 2$, $f(n) = O(1)$.

Depending on the tradeoff between a, b and $f(n)$ different solution to the recurrence.

$$\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a^2T(n/b^2) + (f(n) + af(n/b)) \\
&= \dots \\
&= a^kT(n/b^k) + (f(n) + af(n/b) + \dots a^{k-1}f(n/b^{k-1}) \\
&= \text{ (assuming } n = b^\ell) \; = a^\ell T(1) + (f(b^\ell) + af(b^{\ell-1}) + \dots + a^{\ell-1}f(b)) = \\
&= a^{\log_b n}T(1) + (f(b^\ell) + af(b^{\ell-1}) + \dots + a^{\ell-1}f(b)) = \\
&= \Theta(n^{\log_b a}) + (f(b^\ell) + af(b^{\ell-1}) + \dots + a^{\ell-1}f(b))
\end{aligned}$$

Now what about $f(\cdot)$:

e.g.1 if $f(n) = \Theta(n^{\log_b a})$, easy to show: $a^k f(b^{\ell-k}) = \Theta\left(n^{\log_b a}\right)$, for all k. Hence, we get from the above

$$T(n) = \Theta(n^{\log_b a}\log_b n).$$

e.g.2 if $f(n) = \Theta(n^{\log_b a-\epsilon})$ for some $\epsilon > 0$, easy to show: $a^k f(b^{\ell-k}) = \Theta\left(n^{\log_b a-\epsilon}\cdot b^{k\epsilon}\right)$, for all k. Hence, we get from the above

$$T(n) = \Theta(n^{\log_b a}),$$

because the sum $f(b^\ell) + af(b^{\ell-1}) + \dots + a^{\ell-1}f(b) = n^{\log_b a}$.

e.g.3 if $f(n) = \Theta(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$, easy to show: $a^k f(b^{\ell-k}) = \Theta\left(n^{\log_b a+\epsilon}\cdot b^{-k\epsilon}\right)$, for all k. Hence, we get from the above

$$T(n) = \Theta(n^{\log_b a+\epsilon}),$$

because the sum $f(b^\ell) + af(b^{\ell-1}) + \dots + a^{\ell-1}f(b) = \Theta(n^{\log_b a+\epsilon})$.