Inf 2B: Sorting, MergeSort and Divide-and-Conquer

Lecture 7 of ADS thread

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh

The Sorting Problem

Input:	Array A of *items* with comparable *keys*.
Task:	Sort the items in A by increasing keys.

The number of items to be sorted is usually denoted by n.

What is important?

Worst-case running-time:
What are the bounds on $T_{\text{Sort}}(n)$ for our Sorting Algorithm Sort.

In-place or not?:
A sorting algorithm is *in-place* if it can be (simply) implemented on the input array, with only $O(1)$ extra space (extra variables).

Stable or not?:
A sorting algorithm is *stable* if for every pair of indices with $A[i].key = A[j].key$ and $i < j$, the entry $A[i]$ comes before $A[j]$ in the output array.

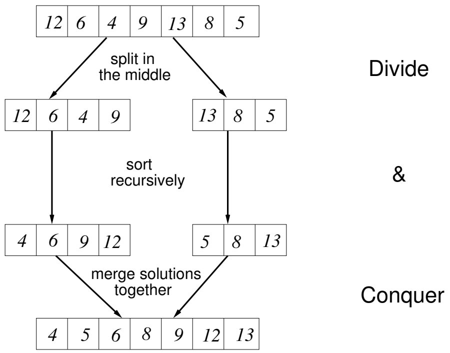
Insertion Sort

Algorithm insertionSort(A)

```
for j ← 1 to A.length − 1 do
    a ← A[j]
    i ← j − 1
    while i ≥ 0 and A[i].key > a.key do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← a
```

- Asymptotic worst-case running time: $\Theta(n^2)$.
- The worst-case (which gives $\Omega(n^2)$) is $\langle n, n-1, \ldots, 1\rangle$.
- *Both* stable and in-place.

2nd sorting algorithm - **Merge Sort**



Merge Sort - recursive structure

Algorithm mergeSort(A, i, j)

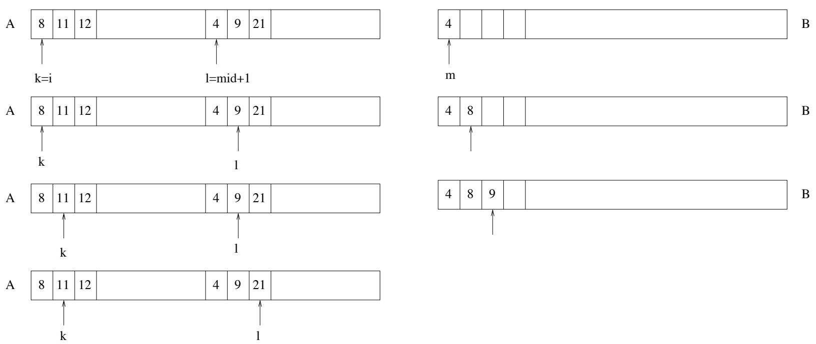
```
if i < j then
    mid ← ⌊(i+j)/2⌋
    mergeSort(A, i, mid)
    mergeSort(A, mid + 1, j)
    merge(A, i, mid, j)
```

Running Time:

$$T(n) = \begin{cases} \Theta(1), & \text{for } n \leq 1; \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + T_{\text{merge}}(n) + \Theta(1), & \text{for } n \geq 2. \end{cases}$$

How do we perform the merging?

Merging the two subarrays



New array B for output.
$\Theta(j - i + 1)$ time (linear time) always (best and worst cases).

Merge pseudocode

Algorithm merge(A, i, mid, j)

```
new array B of length j − i + 1
k ← i
ℓ ← mid + 1
m ← 0
while k ≤ mid and ℓ ≤ j do
    if A[k].key <= A[ℓ].key then
        B[m] ← A[k]
        k ← k + 1
    else
        B[m] ← A[ℓ]
        ℓ ← ℓ + 1
    m ← m + 1
while k ≤ mid do
    B[m] ← A[k]
    k ← k + 1
    m ← m + 1
while ℓ ≤ j do
    B[m] ← A[ℓ]
    ℓ ← ℓ + 1
    m ← m + 1
for m = 0 to j − i do
    A[m + i] ← B[m]
```

Question on mergeSort

What is the status of mergeSort in regard to *stability* and *in-place sorting*?

1. *Both* stable and in-place.
2. Stable but *not* in-place.
3. *Not* stable, but *is* in-place.
4. *Neither* stable nor in-place.

Answer: *not* in-place but it is stable.

If line 6 reads $<$ instead of $<=$, we have sorting but NOT Stability.

Analysis of Mergesort

- **merge**

$$T_{\text{merge}}(n) = \Theta(n)$$

- **mergeSort**

$$\begin{aligned} T(n) &= \begin{cases} \Theta(1), & \text{for } n \leq 1; \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + T_{\text{merge}}(n) + \Theta(1), & \text{for } n \geq 2. \end{cases} \\ &= \begin{cases} \Theta(1), & \text{for } n \leq 1; \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & \text{for } n \geq 2. \end{cases} \end{aligned}$$

Solution to recurrence:

$$T(n) = \Theta(n \lg n).$$

Solving the mergeSort recurrence

Write with constants c, d:

$$T(n) = \begin{cases} c, & \text{for } n \leq 1; \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + dn, & \text{for } n \geq 2. \end{cases}$$

Suppose $n = 2^k$ for some k. Then no floors/ceilings.

$$T(n) = \begin{cases} c, & \text{for } n = 1; \\ 2T(\frac{n}{2}) + dn, & \text{for } n \geq 2. \end{cases}$$

Solving the mergeSort recurrence

Put $\ell = \lg n$ (hence $2^\ell = n$).

$$\begin{aligned} T(n) &= 2T(n/2) + dn \\ &= 2(2T(n/2^2) + d(n/2)) + dn \\ &= 2^2T(n/2^2) + 2dn \\ &= 2^2(2T(n/2^3) + d(n/2^2)) + 2dn \\ &= 2^3T(n/2^3) + 3dn \\ &\ \ \vdots \\ &= 2^kT(n/2^k) + kdn \\ &= 2^\ell T(n/2^\ell) + \ell dn \\ &= nT(1) + \ell dn \\ &= cn + dn \lg(n) \\ &= \Theta(n \lg(n)). \end{aligned}$$

Can extend to n not a power of 2 (see notes).

Merge Sort vs. Insertion Sort

- Merge Sort is much more efficient

But:

- If the array is “almost” sorted, Insertion Sort only needs “almost” linear time, while Merge Sort needs time $\Theta(n \lg(n))$ even in the best case.
- For very small arrays, Insertion Sort is better because Merge Sort has overhead from the recursive calls.
- Insertion Sort sorts **in place**, mergeSort does not (needs $\Omega(n)$ additional memory cells).

Divide-and-Conquer Algorithms

- Divide the input instance into several instances $P_1, P_2, \ldots P_a$ of the same problem of smaller size - “setting-up".
- Recursively solve the problem on these smaller instances.
  - Solve small enough instances directly.
- Combine the solutions for the smaller instances $P_1, P_2, \ldots P_a$ to a solution for the original instance. Do some “extra work" for this.

Analysing Divide-and-Conquer Algorithms

Analysis of divide-and-conquer algorithms yields recurrences like this:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < n_0; \\ T(n_1) + \ldots + T(n_a) + f(n), & \text{if } n \geq n_0. \end{cases}$$

$f(n)$ is the time for “setting-up" and “extra work."

Usually recurrences can be simplified:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < n_0; \\ aT(n/b) + \Theta(n^k), & \text{if } n \geq n_0, \end{cases}$$

where $n_0, a, k \in \mathbb{N}$, $b \in \mathbb{R}$ with $n_0 > 0$, $a > 0$ and $b > 1$ are constants.
(Disregarding floors and ceilings.)

The Master Theorem

Theorem: Let $n_0 \in \mathbb{N}$, $k \in \mathbb{N}_0$ and $a, b \in \mathbb{R}$ with $a > 0$ and $b > 1$, and let $T : \mathbb{N} \to \mathbb{R}$ satisfy the following recurrence:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n < n_0; \\ aT(n/b) + \Theta(n^k), & \text{if } n \geq n_0. \end{cases}$$

Let $e = \log_b(a)$; we call e the *critical exponent*. Then

$$T(n) = \begin{cases} \Theta(n^e), & \text{if } k < e & \textbf{(I)}; \\ \Theta(n^e \lg(n)), & \text{if } k = e & \textbf{(II)}; \\ \Theta(n^k), & \text{if } k > e & \textbf{(III)}. \end{cases}$$

- Theorem still true if we replace $aT(n/b)$ by

$$a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil)$$

for $a_1, a_2 \geq 0$ with $a_1 + a_2 = a$.

Master Theorem in use

Example 1:
We can “read off” the recurrence for mergeSort:

$$T_{\text{mergeSort}}(n) = \begin{cases} \Theta(1), & n \le 1; \\ T_{\text{mergeSort}}(\lceil \frac{n}{2} \rceil) + T_{\text{mergeSort}}(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & n \ge 2. \end{cases}$$

In Master Theorem terms, we have

$$n_0 = 2, \quad k = 1, \quad a = 2, \quad b = 2.$$

Thus

$$e = \log_b(a) = \log_2(2) = 1.$$

Hence

$$T_{\text{mergeSort}}(n) = \Theta(n \lg(n))$$

by case (II).

Further Reading

- If you have [GT], the “Sorting Sets and Selection" chapter has a section on mergeSort(.)
- If you have [CLRS], there is an entire chapter on recurrences.

. . . **Master Theorem**

Example 2: Let T be a function satisfying

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \le 1; \\ 7T(n/2) + \Theta(n^4), & \text{if } n \ge 2. \end{cases}$$

$$e = \log_b(a) = \log_2(7) < 3$$

So $T(n) = \Theta(n^4)$ by case (III) .