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 $T_{Sort}(n)$ 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?:
Algorithm insertionSort(A)

1. for $j \leftarrow 1$ to $A.length - 1$ do
2.     $a \leftarrow A[j]$
3.     $i \leftarrow j - 1$
4.     while $i \geq 0$ and $A[i].key > a.key$ do
5.         $A[i + 1] \leftarrow A[i]$
6.         $i \leftarrow i - 1$
7.     $A[i + 1] \leftarrow a$

- 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.
Second sorting algorithm - Merge Sort

Divide & Conquer

Split in the middle

Sort recursively

Merge solutions together

12 6 4 9 13 8 5

4 6 9 12 5 8 13

4 5 6 8 9 12 13
**Merge Sort** - recursive structure

**Algorithm** `mergeSort(A, i, j)`

1. if $i < j$ then
2.   $mid \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
3.   `mergeSort(A, i, mid)`
4.   `mergeSort(A, mid + 1, j)`
5.   `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$(A, i, mid, j)$

1. new array $B$ of length $j - i + 1$
2. $k \leftarrow i$
3. $\ell \leftarrow mid + 1$
4. $m \leftarrow 0$
5. while $k \leq mid$ and $\ell \leq j$ do
6.     if $A[k].key \leq A[\ell].key$ then
7.         $B[m] \leftarrow A[k]$
8.         $k \leftarrow k + 1$
9.     else
10.        $B[m] \leftarrow A[\ell]$
11.       $\ell \leftarrow \ell + 1$
12.       $m \leftarrow m + 1$
13. while $k \leq mid$ do
14.       $B[m] \leftarrow A[k]$
15.       $k \leftarrow k + 1$
16.       $m \leftarrow m + 1$
17. while $\ell \leq j$ do
18.       $B[m] \leftarrow A[\ell]$
19.       $\ell \leftarrow \ell + 1$
20.       $m \leftarrow m + 1$
21. for $m = 0$ to $j - i$ do
22.       $A[m + i] \leftarrow B[m]$
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

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

Solution to recurrence:

\[ T(n) = \Theta(n \log n). \]
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 $2^\ell = n$).

\[
T(n) = 2T(n/2) + dn
= 2(2T(n/2^2) + d(n/2)) + dn
= 2^2 T(n/2^2) + 2dn
= 2^2 (2T(n/2^3) + d(n/2^2)) + 2dn
= 2^3 T(n/2^3) + 3dn
\]
\[\vdots\]
\[
= 2^k T(n/2^k) + kdn
= 2^\ell T(n/2^\ell) + \ell dn
= nT(1) + \ell dn
= cn + dn \lg(n)
= \Theta(n \lg(n)).
\]

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 $T : \mathbb{N} \rightarrow \mathbb{R}$ 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

\[
T(n) = \begin{cases} 
\Theta(n^e), & \text{if } k < e \quad (\text{I}); \\
\Theta(n^e \log(n)), & \text{if } k = e \quad (\text{II}); \\
\Theta(n^k), & \text{if } k > e \quad (\text{III}).
\end{cases}
\]

▶ 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:

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

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

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

by case (II).
Example 2: Let $T$ be a function satisfying

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

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

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