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 \( A[i].\text{key} = A[j].\text{key} \) and \( i < j \), the entry \( A[i] \) comes before \( A[j] \) in the output array.

---

**Insertion Sort**

**Algorithm** \( \text{insertionSort}(A) \)

1. \( \text{for } j \leftarrow 1 \text{ to } A.\text{length} - 1 \text{ do} \)
2. \( \quad a \leftarrow A[j] \)
3. \( \quad i \leftarrow j - 1 \)
4. \( \quad \text{while } i \geq 0 \text{ and } A[i].\text{key} > a.\text{key} \text{ do} \)
5. \( \quad \qquad A[i + 1] \leftarrow A[i] \)
6. \( \quad \qquad i \leftarrow i - 1 \)
7. \( \quad A[i + 1] \leftarrow a \)

- 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.
2nd sorting algorithm - **Merge Sort**

Merge Sort - recursive structure

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

1. if $i < j$ then
2.    \[ mid \leftarrow \left\lceil \frac{i+j}{2} \right\rceil \]
3.    mergeSort($A, i, mid$)
4.    mergeSort($A, mid + 1, j$)
5.    merge($A, i, mid, j$)

Running Time:

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

How do we perform the merging?

**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.        $B[m] \leftarrow A[k]$ \[ k \leftarrow k + 1 \]
8.    else
9.        $B[m] \leftarrow A[\ell]$ \[ \ell \leftarrow \ell + 1 \]
10.   \[ m \leftarrow m + 1 \]
11. end while
12. while $\ell \leq j$ do
13.    $A[m+i] \leftarrow B[m]$ \[ m \leftarrow m + 1 \]
14.    \[ \ell \leftarrow \ell + 1 \]
15. end while
16. \[ k \leftarrow k + 1 \]
17. while $k \leq mid$ do
18.    $B[m] \leftarrow A[k]$ \[ m \leftarrow m + 1 \]
19.    $k \leftarrow k + 1$ \[ \ell \leftarrow \ell + 1 \]
20. end while
21. \[ B[m] \leftarrow A[k] \]
22. \[ m \leftarrow m + 1 \]
23. \[ \ell \leftarrow \ell + 1 \]
24. \[ k \leftarrow k + 1 \]
25. end while

Merging the two subarrays

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

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**

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

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

Solution to recurrence:

\[ T(n) = \Theta(n \log n). \]

Solving the mergeSort recurrence

Write with constants \(c, d\):

\[ T(n) = \begin{cases} 
  c, & \text{for } n \leq 1; \\
  2T\left(\lfloor \frac{n}{2} \rfloor \right) + dn, & \text{for } n \geq 2.
  \end{cases} \]

Suppose \( n = 2^k \) for some \( k \). Then no floors/ceilings.

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

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.

(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

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

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

$$a_1T(\lfloor n/b \rfloor) + a_2T(\lceil n/b \rceil)$$

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).

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

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

Thus

\[
e = \log_b(a) = \log_7(7) < 3
\]

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