A Schema for Sorting

// Sort input set S
B = empty sequence;
while (S is not empty) {
    x = minimum of S;
    remove x from S;
    put x at the end of B;
}

Why it works:
1. In an iteration, the multiset S ∪ B does not change, as we transfer x from S to B.
2. No entry of B is larger than an element of S.
3. Entries of B are in non-decreasing order.

Thus when this algorithm terminates, all initial elements of S are transferred to B and B is sorted.

Remark.

The operation performed in lines marked © is denoted \( x = \text{deletemin}(S) \).
This schema is used in several algorithms, Heap Sort, Bubble Sort and Merging.
Example:

<table>
<thead>
<tr>
<th>Sequence B</th>
<th>input S</th>
<th>deletemin</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>{12,7,22,5,10,3}</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>{12,7,22,5,10}</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>{12,7,22,5,10}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{12,7,22,10}</td>
<td>5</td>
</tr>
<tr>
<td>3,5</td>
<td>{12,7,22,10}</td>
<td></td>
</tr>
<tr>
<td>3,5</td>
<td>{12,22,10}</td>
<td>7</td>
</tr>
<tr>
<td>3,5,7</td>
<td>{12,22,10}</td>
<td></td>
</tr>
<tr>
<td>3,5,7</td>
<td>{22,10}</td>
<td>12</td>
</tr>
<tr>
<td>3,5,7,12</td>
<td>{22,10}</td>
<td></td>
</tr>
<tr>
<td>3,5,7,12</td>
<td>{22}</td>
<td>10</td>
</tr>
<tr>
<td>3,5,7,12,10</td>
<td>{22}</td>
<td></td>
</tr>
<tr>
<td>3,5,7,12,10</td>
<td>{}</td>
<td>22</td>
</tr>
<tr>
<td>3,5,7,12,10,22</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>
Convention

Array fragment $A[i], A[i+1], \ldots, A[j-1]$ will be denoted $< A, i, j >$.

Array fragment is a useful piece in a larger data structure.

Test that the fragment is not empty (false otherwise):

$$i < j$$

Removing and returning the first entry of $< A, i, j >$:

$$A[i++]$$

Placing $x$ at the end of fragment $< B, m, n >$:

$$B[n++] = x;$$
Merging

We consider one of the cases when it is easy to perform

\[ x = \text{deletemin}(S) \]

Assume that \( S \) consists of two non-overlapping sorted fragments of array \( A \), namely \( \langle A, i, j \rangle \) and \( \langle A, k, l \rangle \).

One can see that the minimum of \( S \) is one of these two entries: \( A[i] \) or \( A[k] \). Thus we can perform \( \text{deletemin}(S) \) as follows:


This expression returns the minimum, and it removes this minimum from the respective fragment. It is correct if both fragments are sorted with the minimum in front, and both are nonempty.

We can implement sorting by \( \text{deletemin} \) as follows:

```c
// merge \langle A, i, j \rangle and \langle A, k, l \rangle into B < m, ?>
int n = m;
while (i < j && k < l)
while (i < j)
    B[n++] = A[i++];
while (k < l)
    B[n++] = A[k++];
```

We say that the above code merges two sorted array fragments.

One can see that for every entry of the resulting sorted fragment we perform only constant number of steps (i.e. the number that is bounded by some constant that does not depend on \( i, j, k, l \) and \( m \)).
Doubling the size of sorted fragments

Suppose that source array $< S,0,n >$ consists of sorted fragments,
$< S,0,\text{size} >$, $< S,\text{size},2\times \text{size} >$, $< S,2\times \text{size},3\times \text{size} >$, etc.
and perhaps one final shorter sorted fragment, $< S,k\times \text{size},n >$.

We can perform a series of mergings to move all entries of $< S,0,n >$ into target $< T,0,n >$
so that target consists of sorted fragments of twice larger size.

```
assumption: every $< A,i\times \text{size},(i+1)\times \text{size} >$ is sorted
i = 0;
while (i < n) {
    k = i+size;
    m = \text{min}(k+size,n);
    j = \text{min}(k,n);
    merge $< S,i,j >$ and $< S,k,m >$ into $< T,i,m >$;
}
achieved: every $< A,2i\times \text{size},(2i+2)\times \text{size} >$ is sorted
```

For every entry of $< T,0,n >$ we perform only a constant number of steps, so we perform $O(n)$ steps.

Now suppose that input is $< A,0,n >$ and that we also have an auxiliary array $< B,0,n >$.
Then we can run the following:

```
int *S, *T, *temp; size,
S = A, T = B;
for (size = 1; size < n; size *= 2) {
    double the size of sorted fragment;
    temp = S, S = T, T = temp;
}
```

At the end, one can see that $< S,0,n >$ stores the output of sorting problem.
Each run of the for loop takes $O(n)$ steps.
One can see that we have exactly $\lceil \log_2 n \rceil$ runs, thus we sort in $O(n \log_2 n)$ steps.
Putting everything together:

```c
int *S, *T, *temp; size,
S = A, T = B;
for (size = 1; size < n; size *= 2) {
    for (i = 0; i < n; ) {
        j = i;
        k = i+size;
        l = k;
        m = k+size < n? k+size : n;
        while (j < k && l < m)
        while (l < m)
            T[i++] = S[l++];
        while (i < m)
            B[i++] = A[j++];
    }
    temp = S, S = T, T = temp;
}
```

Regular set-up for merging fragments:

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>i+size</th>
<th>i+2×size</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>j</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>T</td>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First special set-up for merging:

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>i+size</th>
<th>i+2×size</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>j</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>T</td>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Second special set-up for merging:

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>n</th>
<th>i+size</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>j</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>m</td>
<td>S</td>
<td>l</td>
</tr>
<tr>
<td>T</td>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


**Divide and conquer**

Another way of looking at Merge Sort is that it is a case of applying an important technique of algorithm design called *divide and conquer*.

A problem has **instances**.

Every instance has a **size**.

Small instances are **elementary**, i.e. easy to solve.

To solve a non-elementary instance $S$ of a certain size, say $n$, we
- create a number of smaller instances, $S_1, \ldots, S_k$ with sizes $n_1, \ldots, n_k$ (divide).
- solve $S_1, \ldots, S_k$.
- combine the solutions (conquer).

Suppose that the running time needed to create the smaller instances and to combine the solutions is $t(n)$. Then the running time of this divide and conquer algorithm can be estimate with this recurrence relation:

$$T(n) = t(n) + T(n_1) + \cdots + T(n_k).$$
In our case,

- an instance is a set to be sorted;
- the size is the number of elements;
- we divide set \( S \) into \( S_1 \cup S_2 \), two parts of roughly equal size;
- it is reasonable to define \( t(n) = n \).

Thus we have a recurrence relation problem:

\[
T(1) = 0; \\
T(n) = n + T\left(\lceil \frac{n}{2} \rceil \right) + T\left(\lfloor \frac{n}{2} \rfloor \right) \text{ for } N > 1.
\]

First, we solve this pretending that \( n/2 \) is always integer, and thus rounding operations do not change any values. By induction, we will show that \( T(n) = n \log_2 n \). The basis is true because \( \log_2 1 = 0 \). Inductive step can be shown as follows:

\[
T(n) = n + 2T(\frac{n}{2}) = \\
n + 2(\frac{n}{2} \log_2(\frac{n}{2})) = \\
n + n (\log_2 n + \log_2 \frac{1}{2}) = \\
n + n(\log_2 n - 1) = n \log_2 n.
\]

If we take the rounding effect into account, the result is more complicated.

We can solve this complicated recurrence relation exactly, or we can use Master Theorem.
Assume that \( n = 2^k + m \), where \( 0 \leq m \leq 2^k \). Then

\[
T(n) = k2^k + (k + 2)m \leq (k + 1)n. \quad \text{If } m > 0, \quad \text{then } k + 1 = \lceil \log_2 n \rceil, \quad \text{thus } T(n) \leq \lceil \log_2 n \rceil n.
\]

The basis of this claim, for \( n = 1, k = m = 0 \) is obvious. First will show the inductive step for odd \( n \).

\[
T(2^k + m) = 2^k + m + T(2^{k-1} + \frac{m-1}{2}) + T(2^{k-1} + \frac{m+1}{2}) =
\]

\[
2^k + m + (k-1)2^{k-1} + (k+1) \frac{m-1}{2} + (k-1)2^{k-1} + (k+1) \frac{m+1}{2} =
\]

\[
(1 + \frac{k-1}{2} + \frac{k-1}{2})2^k + (1 + \frac{k+1}{2} + \frac{k+1}{2})m + (k+1)(-\frac{1}{2} + \frac{1}{2}) =
\]

\[
k2^k + (k+2)m.
\]

The case of even \( n \) is simpler:

\[
T(2^k + m) = 2^k + m + 2T(2^{k-1} + \frac{m}{2}) =
\]

\[
2^k + m + 2((k-1)2^{k-1} + (k-1) \frac{m}{2}) =
\]

\[
2^k + m + ((k-1)2^k + (k-1)m) =
\]

\[
k2^k + (k+2)m.
\]