The Template Method Pattern
Used when a system must implement an algorithm

- with an invariant part, which always executes the same way,

- but which depends on variant parts that are implemented differently depending on the circumstance.
Two kinds of methods in the Template Method Pattern:

- The template method
- Primitive operations
Design Patterns for Data Structures

Figure 4.6

```
ASorter
+ sort(a: ASeq<T>&, lo: int, hi: int)
# split(a: ASeq<T>&, lo: int, mid: int&, hi: int)
# join(a: ASeq<T>&, lo: int, mid: int, hi: int)
```

```
MergeSorter
- _tempA: ArrayT<T>
+ MergeSorter (cap: int)
  # split (...)
  # join(...)
```

```
QuickSorter
# split (...)
# join(...)
```

```
SelectSorter
# split (...)
# join(...)
```

```
HeapSorter
+ HeapSorter (a: ASeq<T>&, lo: int, hi: int)
# split (...)
# join(...)
template<class T>
class ASorter {

public:
    void sort(ASeq<T> &a, int lo, int hi);
    // Pre: 0 <= lo && hi < a.cap().
    // Post: sorted(a[lo..hi]).

    virtual ~ASorter() = default;
    // Virtual destructor necessary for subclassing.

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) = 0;
    // Pre: lo < hi.
    // Post: lo < mid <= hi.

    virtual void join(ASeq<T> &a, int lo, int mid, int hi) = 0;
    // Pre: lo < mid <= hi.
    // Pre: sorted(a[lo..mid-1]) && sorted(a[mid..hi]).
    // Post: sorted(a[lo..hi]).
};
template<class T>
void ASorter<T>::sort(ASeq<T> &a, int lo, int hi) {
    if (lo < 0 || a.cap() <= hi) {
        cerr << "ASorter<T>::sort precondition failed." << endl;
        cerr << "lo == " << lo << " a.cap() == " << a.cap() << " hi == " << hi << endl;
        throw -1;
    }
    if (lo < hi) {
        int mid;
        split(a, lo, mid, hi);
        sort(a, lo, mid - 1);
        sort(a, mid, hi);
        join(a, lo, mid, hi);
    }
}
template<class T>
void ASorter<T>::sort(ASeq<T> &a, int lo, int hi) {
    if (lo < 0 || a.cap() <= hi) {
        cerr << "ASorter<T>::sort precondition failed." << endl;
        cerr << "lo == " << lo << " a.cap() == " << a.cap()
             << " hi == " << hi << endl;
        throw -1;
    }
    if (lo < hi) {
        int mid;
        split(a, lo, mid, hi);
        sort(a, lo, mid - 1);
        sort(a, mid, hi);
        join(a, lo, mid, hi);
    }
}

Figure 4.7
Sort precondition:
\[0 \leq l \land h < a.cap()\]

\texttt{sort(a, lo, hi);}

Sort postcondition:
\texttt{sorted(a[\ldots h])}
if $l < h$

Sort precondition:
$0 \leq l \land h < a.cap()$

sort(a, lo, hi);

Sort postcondition:
sorted(a[l..h])
Sort precondition:
$0 \leq l \land h < a.cap()$

$\text{sort}(a, \text{lo}, \text{hi});$

Split precondition:
$l < h$

if $l < h$

$\text{split}(a, \text{lo}, \text{mid}, \text{hi});$

Split postcondition:
$l < m \leq h$

Sort postcondition:
$\text{sorted}(a[l..h])$
Sort precondition:
\[0 \leq l \land h < a.cap()\]
\[\text{sort}(a, \text{lo}, \text{hi});\]

Sort postcondition:
\[\text{sorted}(a[l..h])\]

Split precondition:
\[l < h\]
\[\text{split}(a, \text{lo}, \text{mid}, \text{hi});\]

Split postcondition:
\[l < m \leq h\]
\[\text{sort}(a, \text{lo}, \text{mid} - 1);\]
\[\text{sort}(a, \text{mid}, \text{hi});\]
Split precondition:
\( l < h \)

Split postcondition:
\( l < m \leq h \)

Sort precondition:
\( 0 \leq l \land h < a.\text{cap}() \)

Sort postcondition:
\( \text{sorted}(a[l..h]) \)

if \( l < h \)

\[ \begin{align*}
\text{if } l < h & \quad \text{split}(a, \text{lo}, \text{mid}, \text{hi}); \\
\text{split postcondition: } & \quad \text{sorted}(a[l..h]) \\
\text{Join precondition: } & \quad l < m \leq h \land \text{sorted}(a[l..m-1]) \land \text{sorted}(a[m..h]) \\
\text{Join postcondition: } & \quad \text{sorted}(a[l..h])
\end{align*} \]
Design Patterns for Data Structures

Figure 4.7

Sort precondition: 
$0 \leq l \land h < a \cdot \text{cap}()$

sort($a$, $lo$, $hi$);

Sort postcondition: 
sorted($a[l..h]$)

if $l < h$

Split precondition:
$l < h$

split($a$, $lo$, $mid$, $hi$);

Split postcondition:
$l < m \leq h$

sort($a$, $lo$, $mid - 1$);

sort($a$, $mid$, $hi$);

Join precondition:
$l < m \leq h \land \text{sorted}(a[l..m-1]) \land \text{sorted}(a[m..h])$

join($a$, $lo$, $mid$, $hi$);

Join postcondition:
sorted($a[l..h]$)
The \texttt{split()} primitive operation

(a) The general array $a$.

(b) Array $a$ with the smallest segments.

```c
virtual void split(ASeq<T> &a, int lo, int &mid, int hi) = 0;
// Pre: $lo < hi$.
// Post: $lo < mid \leq hi$.
```
Using a sorter

```cpp
shared_ptr<ASorter<int>> sorter;
```
Using a sorter

```cpp
shared_ptr<ASorter<int>> sorter;

sorter = make_shared<QuickSorter<int>>();
```
Using a sorter

shared_ptr<ASorter<int>> sorter;

sorter = make_shared<QuickSorter<int>>();

ArrayT<int> array(promptIntGE("Enter array capacity", 1));
Using a sorter

```cpp
shared_ptr<ASorter<int>> sorter;

sorter = make_shared<QuickSorter<int>>();

ArrayT<int> array(promptIntGE("Enter array capacity", 1));

sorter->sort(array, 0, length - 1);
```
MergeSorter
template<class T>
class MergeSorter : public ASorter<T> {
private:
    ArrayT<T> _tempA;

public:
    MergeSorter(int cap);

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};
template<class T>
MergeSorter<T>::MergeSorter(int cap):
    _tempA(cap) {
}

template<class T>
void MergeSorter<T>::split(ASeq<T> &, int lo, int &mid, int hi) {
    // Post: mid == (lo + hi + 1) / 2
    mid = (lo + hi + 1) / 2;
}

template<class T>
void MergeSorter<T>::join(ASeq<T> &a, int lo, int mid, int hi) {
    cerr << "MergeSorter<T>::join: Exercise for the student." << endl;
    throw -1;
}
### MergeSorter

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>

(a) Initial list
(a) Initial list

(b) split(a, lo, mid, hi)

MergeSorter

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Initial list</td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>(b) split(a, lo, mid, hi)</td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>
MergeSorter

(a) Initial list

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>

(b) split(a, lo, mid, hi)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>

(c) sort(a, lo, mid - 1)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>
### MergeSorter

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Initial list</td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>(b) <code>split(a, lo, mid, hi)</code></td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>(c) <code>sort(a, lo, mid - 1)</code></td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>(d) <code>sort(a, mid, hi)</code></td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>10</td>
<td>30</td>
<td>60</td>
<td>70</td>
<td>95</td>
</tr>
</tbody>
</table>
**MergeSorter**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>

(a) Initial list

(b) `split(a, lo, mid, hi)`

(c) `sort(a, lo, mid - 1)`

(d) `sort(a, mid, hi)`

(e) `join(a, lo, mid, hi)`
(a) Initialize $i$ to the first element of the left segment, $j$ to the first element of the right segment, and $k$ to the first element of the segment in the temporary array.
(a) Initialize $i$ to the first element of the left segment, $j$ to the first element of the right segment, and $k$ to the first element of the segment in the temporary array.

(b) Compare 20 and 10. Because 10 is less than 20, copy $a[j]$ to $\text{tempA}[k]$ and increment $j$. 
(c) Compare 20 and 30. Because 20 is less than 30, copy \( a[i] \) to \( \text{tempA}[k] \) and increment \( i \).
(c) Compare 20 and 30. Because 20 is less than 30, copy $a[i]$ to $\text{tempA}[k]$ and increment $i$.

(d) Because $i$ has increased past the last element of the left segment, do not compare two elements. Copy $a[j]$ to $\text{tempA}[k]$ and increment $j$. 
(b) Compare 20 and 10. Because 10 is less than 20, copy $a[j]$ to $\_\text{tempA}[k]$ and increment $j$.

(a) Initialize $i$ to the first element of the left segment, $j$ to the first element of the right segment, and $k$ to the first element of the segment in the temporary array.

20 40 50 80 90 10 30 60 70 95

i j k

(c) Compare 20 and 30. Because 20 is less than 30, copy $a[i]$ to $\_\text{tempA}[k]$ and increment $i$.

20 40 50 80 90 10 30 60 70 95

i j k

(d) Because $i$ has increased past the last element of the left segment, do not compare two elements. Copy $a[j]$ to $\_\text{tempA}[k]$ and increment $j$.

20 40 50 80 90 10 30 60 70 95

i j k

10 20 30 40 50 60 70 80 90 95

(e) Copy the segment from $\_\text{tempA}$ back into array $a$.
The call tree for MergeSorter

![Call tree diagram for MergeSorter](Image)
The performance of MergeSorter
The performance of MergeSorter

```plaintext
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```
The performance of MergeSorter

```plaintext
sort(a, lo, hi)
  if (lo < hi)
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
    sort(a, mid, hi)
    join(a, lo, mid, hi)
```

What is the recurrence?
The performance of MergeSorter

\[
\text{sort}(a, \text{lo}, \text{hi}) \\
\quad \text{if } (\text{lo} < \text{hi}) \\
\quad \quad \text{split}(a, \text{lo}, \text{mid}, \text{hi}) \\
\quad \quad \text{sort}(a, \text{lo}, \text{mid} - 1) \\
\quad \quad \text{sort}(a, \text{mid}, \text{hi}) \\
\quad \quad \text{join}(a, \text{lo}, \text{mid}, \text{hi}) \\
\]

What is the recurrence?

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1. 
\end{cases} 
\]
Solution to the recurrence

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

is

\[ T(n) = \Theta(n \lg n) \]
Demo SortInt Project
Quick Sort

(a) The merge sort algorithm.

(b) The quick sort algorithm.
Quick Sort

What value should you use for key in the split?

(b) The quick sort algorithm.
Quick Sort

What value should you use for `key` in the split?

The best value would be the median.

(b) The quick sort algorithm.
Quick Sort

What value should you use for key in the split?

The best value would be the median.

But, to find the median, you must sort the list!

(b) The quick sort algorithm.
Quick Sort

Techniques for estimating the median:

• Estimate the median of the elements in \(a[l..h]\) as \(a[h]\).

• Estimate the median of the elements in \(a[l..h]\) as \(a[r]\), where \(r\) is an integer index chosen at random from the range \(l..h\).

• Estimate the median of the elements in \(a[l..h]\) as the median of the three values \(a[r], a[s],\) and \(a[t]\) where \(r, s,\) and \(t\) are integer indices chosen at random from the range \(l..h\).
QuickSorter

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th></th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>

(a) Initial list
Design Patterns for Data Structures

**Figure 4.13**

QuickSorter

(a) Initial list

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
</tr>
</tbody>
</table>

(b) `split(a, lo, mid, hi)`

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>30</td>
<td>60</td>
<td>95</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>
QuickSorter

(a) Initial list

(b) `split(a, lo, mid, hi)`

(c) `sort(a, lo, mid - 1)`
QuickSorter

(a) Initial list

(b) split(a, lo, mid, hi)

(c) sort(a, lo, mid - 1)

(d) sort(a, mid, hi)
Design Patterns for Data Structures

QuickSorter

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[8]</td>
<td>[9]</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>95</td>
<td>60</td>
<td>30</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

(b) $\text{split}(a, \text{lo}, \text{mid}, \text{hi})$

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>50</td>
<td>40</td>
<td>10</td>
<td>30</td>
<td>60</td>
<td>95</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

(c) $\text{sort}(a, \text{lo}, \text{mid} - 1)$

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>95</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

(d) $\text{sort}(a, \text{mid}, \text{hi})$

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>

(e) $\text{join}(a, \text{lo}, \text{mid}, \text{hi})$
QuickSorter

(a) Initial list.

90  20  80  50  40  10  95  60  30  70

(b) Take a random sample of three elements. Swap the median of these three elements with a[hi].

key=a[hi], which is 60.

(c) Initialize mid = lo, j = lo.

(d) Compare 90 and 60. Because 90 is greater than 60, only increment j.

(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both j and mid.

(f) Compare 80 and 60. Because 80 is greater than 60, only increment j.

(g) Compare 50 and 60. Because 50 is less than or equal to 60, swap 50 and 90, and increment both j and mid.

(h) Compare 40 and 60. Because 40 is less than or equal to 60, swap 40 and 80, and increment both j and mid.

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both j and mid.

(j) Compare 95 and 60. Because 95 is greater than 60, only increment j.
QuickSorter

(a) Initial list.

(b) Take a random sample of three elements. Swap the median of these three elements with \(a[hi]\). \text{key}=a[hi], which is 60.
QuickSorter

(a) Initial list.

(b) Take a random sample of three elements. Swap the median of these three elements with $a[hi]$. key=$a[hi]$, which is 60.

(c) Initialize $mid = lo$, $j = lo$. 

(d) Compare 90 and 60. Because 90 is greater than 60, only increment $j$.

(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both $j$ and $mid$.

(f) Compare 80 and 60. Because 80 is greater than 60, only increment $j$.

(g) Compare 50 and 60. Because 50 is less than or equal to 60, swap 50 and 90, and increment both $j$ and $mid$.

(h) Compare 40 and 60. Because 40 is less than or equal to 60, swap 40 and 80, and increment both $j$ and $mid$.

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both $j$ and $mid$.

(j) Compare 95 and 60. Because 95 is greater than 60, only increment $j$.
(a) Initial list.

90  20  80  50  40  10  95  60  30  70

(b) Take a random sample of three elements. Swap the median of these three elements with $a[hi]$. key=$a[hi]$, which is 60.

(c) Initialize mid = lo, j = lo.

(d) Compare 90 and 60. Because 90 is greater than 60, only increment j.
(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both $j$ and $mid$. 

QuickSorter
QuickSorter

(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both j and mid.

(f) Compare 80 and 60. Because 80 is greater than 60, only increment j.
(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both j and mid.

(f) Compare 80 and 60. Because 80 is greater than 60, only increment j.

(g) Compare 50 and 60. Because 50 is less than or equal to 60, swap 50 and 90, and increment both j and mid.
(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both j and mid.

(f) Compare 80 and 60. Because 80 is greater than 60, only increment j.

(g) Compare 50 and 60. Because 50 is less than or equal to 60, swap 50 and 90, and increment both j and mid.

(h) Compare 40 and 60. Because 40 is less than or equal to 60, swap 40 and 80, and increment both j and mid.
QuickSorter

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both j and mid.
Figure 4.14  

QuickSorter

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both \( j \) and \( \text{mid} \).

(j) Compare 95 and 60. Because 95 is greater than 60, only increment \( j \).  

...
QuickSorter

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both j and mid.

(j) Compare 95 and 60. Because 95 is greater than 60, only increment j.

(k) Compare 60 and 60. Because 60 is less than or equal to 60, swap 60 and 90, and increment mid.
```cpp
#include <random>
#include "ASorter.hpp"

template<class T>
class QuickSorter : public ASorter<T> {

protected:
virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;

private:
    random_device rdev{};
    default_random_engine engine{rdev()};
};
```

Figure 4.15

QuickSorter.hpp

Implementaiton of the quicksort algorithm. The program listing continues in the next figure.

As a program will execute at a different time, and the seed value returned by `rdev` will be different. Some processors have a sensor that measures a signal from some random physical event. Whatever device `rdev` is tied to, it is impossible to predict what seed value it will provide. Once you set the seed to an initial value, subsequent random numbers are determined precisely by the generating algorithm and could theoretically be predicted. Because the algorithm is usually complex, it is difficult in practice to make such predictions even if you know the initial seed value. If you run `QuickSorter` twice, the second run will have a different seed, the random number generator will produce a different sequence of random numbers, and the splits will not be identical to those of the first run.

If you are trying to debug a program that initializes the seed using this technique and you want the sequence of random numbers to be the same from run to other, you can change the initialization to `default_random_engine engine{1};` Because this statement initializes the seed to 1, the same sequence of pseudo-random numbers will be generated with every run.

Figure 4.16 shows the `split()` method for `QuickSorter`. The line `uniform_int_distribution<int> distr(lo, hi);` declares local object `distr`, which calls its constructor with parameters `lo` and `hi`.

Distributions operate in conjunction with engines. An engine provides random values to a distribution, and the distribution filters the values to provide the required distribution of random values. As the name of its class indicates, `distr` provides a uniform distribution of integer values between the values of `lo` and `hi` inclusively. The line `int mdn1 = distr(engine);` calls the overloaded function `distr()` which requires an engine for its parameter and returns a random integer with the required distribution.
template<class T>
void QuickSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
    T temp;
    if (hi - lo > 4) {
        int mdn; // Index of the estimate of the median value.
        uniform_int_distribution<int> distr(lo, hi);
        // Find the estimate of the median.
        int mdn1 = distr(engine);
        int mdn2 = distr(engine);
        int mdn3 = distr(engine);
        if ((a[mdn2] <= a[mdn1] && a[mdn1] <= a[mdn3])
            || (a[mdn3] <= a[mdn1] && a[mdn1] <= a[mdn2])) {
            mdn = mdn1; // a[mdn1] is the median
        } else if ((a[mdn1] <= a[mdn2] && a[mdn2] <= a[mdn3])
            || (a[mdn3] <= a[mdn2] && a[mdn2] <= a[mdn1])) {
            mdn = mdn2; // a[mdn2] is the median
        } else {
            mdn = mdn3; // a[mdn3] is the median
        }
    }
}
Design Patterns for Data Structures

Figure 4.16

// Swap the estimate of the median with a[hi].
temp = a[mdn];
a[mdn] = a[hi];
a[hi] = temp;

// Now do the split.
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
mid = hi < mid ? hi : mid; // If a[hi] contains the maximum element.
template<class T>
void QuickSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
There are two postconditions
There are two postconditions

\[ a[l..m - 1] \leq a[m..h] \]

From `split()` in QuickSorter
There are two postconditions

\[ a[l..m−1] \leq a[m..h] \]

\[ l < m \leq h. \]

From `split()` in QuickSorter

From general `split()`
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
mid = hi < mid ? hi : mid;
Design Patterns for Data Structures

T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
  if (a[j] <= key) {
    temp = a[mid];
    a[mid] = a[j];
    a[j] = temp;
    mid++;
  }
}
mid = hi < mid ? hi : mid;

Guarantees

\[ a[l..m-1] \leq a[m..h] \]
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}

mid = hi < mid ? hi : mid;

Guarantees

\[ a[l..m−1] \leq a[m..h] \]

Guarantees

\[ l < m \leq h. \]
Figure 4.17

(a) The three regions during execution of the loop.

(b) The two regions after normal termination of the loop.
(c) After termination of the loop if \( a[hi] \) contains the maximum element.

(d) After adjustment of \( mid \) if \( a[hi] \) contains the maximum element.
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}

(a) The three regions during execution of the loop.
What is the loop invariant?

(a) The three regions during execution of the loop.
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }

(a) The three regions during execution of the loop.

What is the loop invariant?

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]
(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)

The invariant is true at the beginning of the loop.
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

*Proof:* Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

**Proof**: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle \text{Assignment statements mid} = \text{lo} \text{ and } j = \text{lo} \text{ in sort()} \rangle\]
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

\[ (\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k) \]

\[ = \langle \text{Assignment statements mid} = 10 \text{ and } j = 10 \text{ in sort()}\rangle \]

\[ (\forall i \mid l \leq i < l : a[i] \leq k) \land (\forall i \mid l \leq i < l : a[i] > k) \]
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle \text{Assignment statements mid } = \text{ lo and } j = \text{ lo in sort()} \rangle \]
\[\langle \forall i \mid l \leq i < l : a[i] \leq k \rangle \land (\forall i \mid l \leq i < l : a[i] > k)\]

\[= \langle \text{Math} \rangle\]
Design Patterns for Data Structures

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

**Proof:** Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

=  \langle Assignment statements \text{ mid} = \text{ lo} and \text{ j} = \text{ lo in sort()} \rangle

\[(\forall i \mid l \leq i < l : a[i] \leq k) \land (\forall i \mid l \leq i < l : a[i] > k)\]

=  \langle Math \rangle

\[(\forall i \mid false : a[i] \leq k) \land (\forall i \mid false : a[i] > k)\]
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

**Proof:** Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle\text{Assignment statements mid} = \text{lo and j} = \text{lo in sort( )}\rangle\]
\[\quad (\forall i \mid l \leq i < l : a[i] \leq k) \land (\forall i \mid l \leq i < l : a[i] > k)\]

\[= \langle\text{Math}\rangle\]
\[\quad (\forall i \mid \text{false} : a[i] \leq k) \land (\forall i \mid \text{false} : a[i] > k)\]

\[= \langle\text{Empty range rule}\rangle\]
Design Patterns for Data Structures

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

=  \langle Assignment\ statements\ mid = lo\ and\ j = lo\ in\ sort()\rangle

\[(\forall i \mid l \leq i < l : a[i] \leq k) \land (\forall i \mid l \leq i < l : a[i] > k)\]

=  \langle Math\rangle

\[(\forall i \mid false : a[i] \leq k) \land (\forall i \mid false : a[i] > k)\]

=  \langle Empty\ range\ rule\rangle

true \land true
Design Patterns for Data Structures

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

*Proof*: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle \text{Assignment statements mid} = l0 \text{ and } j = l0 \text{ in sort( )} \rangle\]

\[(\forall i \mid l \leq i < l : a[i] \leq k) \land (\forall i \mid l \leq i < l : a[i] > k)\]

\[= \langle \text{Math} \rangle\]

\[(\forall i \mid false : a[i] \leq k) \land (\forall i \mid false : a[i] > k)\]

\[= \langle \text{Empty range rule} \rangle\]

\[true \land true\]

\[= \langle \text{Idempotency of } \land \rangle\]
Design Patterns for Data Structures

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

The invariant is true at the beginning of the loop.

**Proof**: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle \text{Assignment statements mid} = 10\text{ and } j = 10\text{ in sort()}\rangle\]
\[(\forall i \mid l \leq i < l : a[i] \leq k) \land (\forall i \mid l \leq i < l : a[i] > k)\]

\[= \langle \text{Math} \rangle\]
\[(\forall i \mid false : a[i] \leq k) \land (\forall i \mid false : a[i] > k)\]

\[= \langle \text{Empty range rule} \rangle\]
\[true \land true\]

\[= \langle \text{Idempotency of } \land \rangle\]
\[true\]
The invariant is maintained with each execution of the loop.
The invariant is maintained with each execution of the loop.

See text
The invariant is maintained with each execution of the loop.

See text

The loop terminates.
The invariant is maintained with each execution of the loop.

See text

The loop terminates.

See text
The postcondition holds at the end of the loop.
The postcondition holds at the end of the loop.

**Proof:** Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]
The postcondition holds at the end of the loop.

*Proof:* Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle \text{The final value of } j \text{ is } h + 1 \rangle\]
The postcondition holds at the end of the loop.

*Proof*: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

\[= \langle\text{The final value of } j \text{ is } h + 1\rangle\]

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < h + 1 : a[i] > k)\]
The postcondition holds at the end of the loop.

*Proof:* Starting with the invariant,

\[
(\forall i \mid l \leq i < m \colon a[i] \leq k) \land (\forall i \mid m \leq i < j \colon a[i] > k)
\]

\[
= \langle \text{The final value of } j \text{ is } h + 1 \rangle
\]

\[
(\forall i \mid l \leq i < m \colon a[i] \leq k) \land (\forall i \mid m \leq i < h + 1 \colon a[i] > k)
\]

\[
= \langle \text{Math} \rangle
\]
The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

\[
(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)
\]

\[
= \quad \langle \text{The final value of } j \text{ is } h + 1 \rangle
\]

\[
(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < h + 1 : a[i] > k)
\]

\[
= \quad \langle \text{Math} \rangle
\]

\[
(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i \leq h : a[i] > k)
\]
The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

=  \langle The final value of \( j \) is \( h + 1 \rangle \n
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < h + 1 : a[i] > k)\]

=  \langle Math \rangle \n
\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i \leq h : a[i] > k)\]

\Rightarrow  \langle Transitivity of the < operator and logic \rangle
The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < j : a[i] > k)\]

=  \(\langle\text{The final value of } j \text{ is } h + 1\rangle\)

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i < h + 1 : a[i] > k)\]

=  \(\langle\text{Math}\rangle\)

\[(\forall i \mid l \leq i < m : a[i] \leq k) \land (\forall i \mid m \leq i \leq h : a[i] > k)\]

⇒  \(\langle\text{Transitivity of the } < \text{ operator and logic}\rangle\)

\[a[l..m - 1] \leq a[m..h]\]

which is the second postcondition of \texttt{sort()}. 

Design Patterns for Data Structures
The performance of QuickSorter
The performance of QuickSorter

sort(a, lo, hi)

if (lo < hi)
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
    sort(a, mid, hi)
    join(a, lo, mid, hi)
The performance of QuickSorter

sort(a, lo, hi)
   if (lo < hi)
      split(a, lo, mid, hi)
      sort(a, lo, mid - 1)
      sort(a, mid, hi)
      join(a, lo, mid, hi)

What is the best case recurrence?
The performance of QuickSorter

\[
sort(a, \text{lo, hi})
\]
\[
\quad \text{if (lo < hi)}
\]
\[
\quad \text{split}(a, \text{lo, mid, hi})
\]
\[
\quad \text{sort}(a, \text{lo, mid - 1})
\]
\[
\quad \text{sort}(a, \text{mid, hi})
\]
\[
\quad \text{join}(a, \text{lo, mid, hi})
\]

What is the best case recurrence?

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1. 
\end{cases}
\]
Solution to the best case recurrence

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

is

\[
T(n) = \Theta(n \lg n)
\]
The performance of QuickSorter

```java
sort(a, lo, hi)
if (lo < hi)
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
    sort(a, mid, hi)
    join(a, lo, mid, hi)
```

What is the worst case recurrence?
The performance of QuickSorter

\[
sort(a, \text{lo}, \text{hi}) \\
\text{if} \ (\text{lo} < \text{hi}) \\
\quad \text{split}(a, \text{lo}, \text{mid}, \text{hi}) \\
\quad \text{sort}(a, \text{lo}, \text{mid} - 1) \\
\quad \text{sort}(a, \text{mid}, \text{hi}) \\
\quad \text{join}(a, \text{lo}, \text{mid}, \text{hi})
\]

What is the worst case recurrence?

\[
T(n) = \begin{cases} \\
\Theta(1) & \text{if } n \leq 1, \\
T(n - 1) + \Theta(n) & \text{if } n > 1. \\
\end{cases}
\]
Solution to the worst case recurrence

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
T(n-1) + \Theta(n) & \text{if } n > 1. 
\end{cases} \]

is

\[ T(n) = \Theta(n^2) \]
SelectSorter

template<class T>
class SelectSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};
template<class T>
void SelectSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
    // Post: a[hi] == max(a[lo..hi]).
    // Post: mid == hi.
    int indexOfMax = lo;
    for (int i = lo + 1; i <= hi; i++) {
        if (a[indexOfMax] < a[i]) {
            indexOfMax = i;
        }
    }
    T temp = a[hi];
    a[hi] = a[indexOfMax];
    a[indexOfMax] = temp;
    mid = hi;
}

template<class T>
void SelectSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
template<class T>
void largestLast(ASeq<T> &a, int len) {
    int indexOfMax = 0;
    for (int j = 1; j < len; j++) {
        if (a[indexOfMax] < a[j]) {
            indexOfMax = j;
        }
    }
    T temp = a[len - 1];
    a[len - 1] = a[indexOfMax];
    a[indexOfMax] = temp;
}
Design Patterns for Data Structures

Figure 3.1

(a) Initial list

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(b) $m = 0$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(c) $j = 1$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(d) $j++$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(e) $m = j$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(f) $j++$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(g) $m = j$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(h) $j++$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(i) $j++$;

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>85</td>
<td>40</td>
<td>10</td>
</tr>
</tbody>
</table>

(j) Swap

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
<td>70</td>
<td>10</td>
<td>40</td>
<td>85</td>
</tr>
</tbody>
</table>

$\text{len-1}$
The performance of SelectSorter

\[
\text{sort}(a, \text{lo}, \text{hi})
\]
\[
\text{if } (\text{lo} < \text{hi})
\]
\[
\text{split}(a, \text{lo}, \text{mid}, \text{hi})
\]
\[
\text{sort}(a, \text{lo}, \text{mid} - 1)
\]
\[
\text{sort}(a, \text{mid}, \text{hi})
\]
\[
\text{join}(a, \text{lo}, \text{mid}, \text{hi})
\]

What is the recurrence?
The performance of SelectSorter

\[
\text{sort}(a, \text{lo}, \text{hi}) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
T(n-1) + \Theta(n) & \text{if } n > 1.
\end{cases}
\]

What is the recurrence?
Solution to the recurrence

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
T(n - 1) + \Theta(n) & \text{if } n > 1.
\end{cases}
\]

is

\[
T(n) = \Theta(n^2)
\]
template<class T>
class InsertSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T>&, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T>&, int lo, int mid, int hi) override;
};

Figure 4.19

(a) The insertion sort algorithm.
template<class T>
void InsertSorter<T>::split(ASeq<T> &a, int lo, int mid, int hi) {
  // Post: mid == hi.
  mid = hi;
}

template<class T>
void InsertSorter<T>::join(ASeq<T> &a, int lo, int mid, int hi) {
  // Pre: mid == hi && sorted(a[lo..hi - 1]).
  // Post: sorted(a[lo..hi]).
    cerr << "InsertSorter<T>::join: Exercise for the student."
        << endl;
    throw -1;
}
(a) Initial list.

\[(\begin{array}{cccccccccc}
90 & 20 & 80 & 50 & 40 & 10 & 95 & 60 & 30 & 70 \\
\end{array})\]
(a) Initial list.

(b) `split(a, lo, mid, hi)`
Sets `mid` with `mid = hi`.
(a) Initial list.

(b) split(a, lo, mid, hi)
Sets mid with mid = hi.

(c) sort(a, lo, mid - 1)
(a) Initial list.

(b) `split(a, lo, mid, hi)`
Sets `mid` with `mid = hi`.

(c) `sort(a, lo, mid - 1)`

(d) `sort(a, mid, hi)`
No work is done.
(e) join(a, lo, mid, hi)
Initialize j to mid and key to a[mid], which is 70.
(e) `join(a, lo, mid, hi)`
Initialize `j` to `mid` and `key` to `a[mid]`, which is 70.

(f) Compare 70 and 95. Because 95 is greater than 70, set `a[j]` to 95 and decrement `j`. 
(e) **join**\((a, l_0, m_i_d, h_i)\)
Initialize \(j\) to \(m_i_d\) and \(k_e_y\) to \(a[m_i_d]\), which is 70.

(f) Compare 70 and 95. Because 95 is greater than 70, set \(a[j]\) to 95 and decrement \(j\).

(g) Compare 70 and 90. Because 90 is greater than 70, set \(a[j]\) to 90 and decrement \(j\).
(e) join(a, lo, mid, hi)
Initialize j to mid and key to a[mid], which is 70.

(f) Compare 70 and 95. Because 95 is greater than 70, set a[j] to 95 and decrement j.

(g) Compare 70 and 90. Because 90 is greater than 70, set a[j] to 90 and decrement j.

(h) Compare 70 and 80. Because 80 is greater than 70, set a[j] to 80 and decrement j.
(e) $\text{join}(a, \text{lo}, \text{mid}, \text{hi})$
Initialize $j$ to $\text{mid}$ and $\text{key}$ to $a[\text{mid}]$, which is 70.

(f) Compare 70 and 95. Because 95 is greater than 70, set $a[j]$ to 95 and decrement $j$.

(g) Compare 70 and 90. Because 90 is greater than 70, set $a[j]$ to 90 and decrement $j$.

(h) Compare 70 and 80. Because 80 is greater than 70, set $a[j]$ to 80 and decrement $j$.

(i) Compare 70 and 60. Because 60 is not greater than 70, set $a[j]$ to $\text{key}$ and terminate the loop.
The performance of InsertSorter

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?
The performance of InsertSorter

\[
\text{sort}(a, \text{lo}, \text{hi})
\]
\[
\text{if } (\text{lo} < \text{hi})
\]
\[
\text{split}(a, \text{lo}, \text{mid}, \text{hi})
\]
\[
\text{sort}(a, \text{lo}, \text{mid} - 1)
\]
\[
\text{sort}(a, \text{mid}, \text{hi})
\]
\[
\text{join}(a, \text{lo}, \text{mid}, \text{hi})
\]

What is the recurrence?

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
T(n - 1) + \Theta(n) & \text{if } n > 1.
\end{cases}
\]
Solution to the recurrence

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1, \\
T(n-1) + \Theta(n) & \text{if } n > 1.
\end{cases}
\]

is

\[
T(n) = \Theta(n^2)
\]
Heap Sort

1. **Build heap**
2. **Split**
3. **Join**
4. **Sort**

**Figure 4.5**
Design Patterns for Data Structures

Figure 4.23

\[
\text{maxHeap}(a[l..h]) \equiv \left( \forall i \mid l + 1 \leq i \leq h : a[\lfloor (i + l - 1) / 2 \rfloor] \geq a[i] \right).
\]

\[
\text{maxHeap}(a[l..h]) \equiv \left( \forall i \mid l + 1 \leq i \leq h : a[l] \leq a[i] \right).
\]

\[
\text{maxHeap}(a[l..h]) \equiv \left( \forall i \mid l + 1 \leq i \leq h : \text{max}(a[\lfloor (i + l - 1) / 2 \rfloor], a[i]) = a[i] \right).
\]
• The parent of the node at index \( i \) is at index \( \lfloor (i + l - 1)/2 \rfloor \).
The Template Method Pattern

Figure 4.23

The relation between the index of an element in an array segment and its position in a max-heap. Figure 4.23 shows an example with 90 and 80 as the children of the node at index 7.

- The parent of the node at index $i$ is at index $[(i + l - 1)/2]$.
- The left child of the node at index $i$ is at index $2 \cdot i - l + 1$.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>90</td>
<td>80</td>
<td>60</td>
<td>50</td>
<td>70</td>
<td>10</td>
<td>30</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>
The Template Method Pattern

4.2

Figure 4.23

- The parent of the node at index $i$ is at index $\lfloor(i + l - 1)/2\rfloor$.
- The left child of the node at index $i$ is at index $2 \cdot i - l + 1$.
- The largest index of a node with at least one child is $\lfloor(l + h - 1)/2\rfloor$. 
SiftDown()

Pre: Every node below $i$ satisfies the maxheap property.
Post: Every node including $i$ satisfies the maxheap property.

(a) Initial tree
SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.

(b) Compare 30, 90, 80.
Swap 30 with 90.
Move i to left child.
SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.

(c) Compare 30, 45, 55.
Swap 30 with 55.
Move i to right child.
SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.

(d) Compare 30, 20, 40.
Swap 30 with 40.
Move i to right child.
// ========= siftDown =========
template<class T>
void siftDown(ASeq<T> &a, int lo, int i, int hi) {
    // Pre: maxHeap(a[lo + 1..hi]).
    // Pre: lo <= i <= hi.
    // Post: maxHeap(a[i..hi]).
    int child = 2 * i - lo + 1; // Index of left child.
    if (child <= hi) {
        if (child < hi && a[child] < a[child + 1]) {
            child++;
        } // child is the index of the larger of the two children.
        if (a[i] < a[child]) {
            T temp = a[i];
            a[i] = a[child];
            a[child] = temp;
            siftDown(a, lo, child, hi);
        }
    }
}
The worst case performance of `siftDown`

```java
if (child <= hi) {
    if (child < hi && a[child] < a[child + 1]) {
        child++;
    } // child is the index of the larger of the two children.
    if (a[i] < a[child]) {
        T temp = a[i];
        a[i] = a[child];
        a[child] = temp;
        siftDown(a, lo, child, hi);
    }
}
```
The worst case performance of \texttt{siftDown}

```java
if (child <= hi) {
    if (child < hi && a[child] < a[child + 1]) {
        child++;
    } // child is the index of the larger of the two children.
    if (a[i] < a[child]) {
        T temp = a[i];
        a[i] = a[child];
        a[child] = temp;
        siftDown(a, lo, child, hi);
    }
}
```

Height of heap with n elements
The worst case performance of \texttt{siftDown}

```java
if (child <= hi) {
    if (child < hi && a[child] < a[child + 1]) {
        child++;
    } // child is the index of the larger of the two children.
    if (a[i] < a[child]) {
        T temp = a[i];
        a[i] = a[child];
        a[child] = temp;
        siftDown(a, lo, child, hi);
    }
}
```

Height of heap with \(n\) elements

\[
T(n) = \Theta(\lg n)
\]
(a) The initial list is not a heap.
(a) The initial list is not a heap.

(b) Sift down 6.
(a) The initial list is not a heap.

(b) Sift down 6.

(c) Sift down 1.
(a) The initial list is not a heap.

(b) Sift down 6.

(c) Sift down 1.

(d) Sift down 3.
(a) The initial list is not a heap.

(b) Sift down 6.

(c) Sift down 1.

(d) Sift down 3.

(e) Sift down 7.
template<class T>
class HeapSorter : public ASorter<T> {
public:
    HeapSorter(ASeq<T> &a, int lo, int hi);
    // Constructor initializes a to a heap.
    // Post: maxHeap(a[lo..hi]).

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};

template<class T>
HeapSorter<T>::HeapSorter(ASeq<T> &a, int lo, int hi) {
    // Post: maxHeap(a[lo..hi]).
    for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
        siftDown(a, lo, i, hi);
    }
}
Design Patterns for Data Structures

The worst case performance to build initial heap

```java
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```
The worst case performance to build initial heap

```java
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Loop executes n/2 times.
The worst case performance to build initial heap

```java
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Loop executes n/2 times.

Each time, worst case \( \log n \) executions.
The worst case performance to build initial heap

for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}

Loop executes n/2 times.

Each time, worst case \( lg \, n \) executions.

\[
T(n) = \Theta(n \, \lg \, n)
\]
The worst case performance to build initial heap

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Can show actual performance is

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

Beyond the scope of this book.
(a) The initial heap.
(a) The initial heap. (b) Swap 8 and 3.
Design Patterns for Data Structures

(a) The initial heap.

(b) Swap 8 and 3.

(c) Sift down 3.

Figure 4.27
(a) The initial heap.

(b) Swap 8 and 3.

(c) Sift down 3.

(d) Swap 7 and 3.
Design Patterns for Data Structures

Figure 4.27

(a) The initial heap.

(b) Swap 8 and 3.

(c) Sift down 3.

(d) Swap 7 and 3.

(e) Sift down 3.
(a) The initial heap.

(b) Swap 8 and 3.

(c) Sift down 3.

(d) Swap 7 and 3.

(e) Sift down 3.

(f) Swap 6 and 1.
(g) Sift down 1.
(g) Sift down 1.

(h) Swap 5 and 2.
(g) Sift down 1.

(h) Swap 5 and 2.

(i) Sift down 2.
(j) Swap 4 and 2.
Design Patterns for Data Structures

(j) Swap 4 and 2.

(k) Sift down 2.
Design Patterns for Data Structures

(j) Swap 4 and 2.

(k) Sift down 2.

(l) Swap 3 and 1.
(j) Swap 4 and 2.

(k) Sift down 2.

(l) Swap 3 and 1.

(m) Sift down 1.
(j) Swap 4 and 2.

(k) Sift down 2.

(l) Swap 3 and 1.

(m) Sift down 1.

(n) Swap 2 and 1.
Design Patterns for Data Structures

(j) Swap 4 and 2.

(k) Sift down 2.

(l) Swap 3 and 1.

(m) Sift down 1.

(n) Swap 2 and 1.

(o) Sift down 1.
template<class T>
void HeapSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
    // Pre: maxHeap(a[lo..hi]).
    // Post: maxHeap(a[lo..hi - 1]).
    T temp = a[hi];
    a[hi] = a[lo];
    a[lo] = temp;
    siftDown(a, lo, lo, hi - 1);
    mid = hi;
}

template<class T>
void HeapSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
The performance of `HeapSorter::sort()`

```c
sort(a, lo, hi)
  if (lo < hi)
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
    sort(a, mid, hi)
    join(a, lo, mid, hi)
```
The performance of HeapSorter::sort()

sort(a, lo, hi)
   if (lo < hi)
      split(a, lo, mid, hi)
      sort(a, lo, mid - 1)
      sort(a, mid, hi)
      join(a, lo, mid, hi)

Second sort() does no work.
The performance of `HeapSorter::sort()`

```cpp
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

Second `sort()` does no work.

`join()` does no work.
The performance of HeapSorter::sort()

```cpp
    sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```
The performance of \texttt{HeapSorter::sort()}:

\begin{verbatim}
    sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
\end{verbatim}

What is the time for split?
The performance of `HeapSorter::sort()`

```cpp
sort(a, lo, hi)
if (lo < hi)
    int mid;
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
```

What is the time for split?

$\Theta(\lg n)$
The performance of `HeapSorter::sort()`

```cpp
sort(a, lo, hi)
if (lo < hi)
    int mid;
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
```

What is the time for split?

\[ \Theta(\lg n) \]

How many times does it execute?
The performance of HeapSorter::sort()

```cpp
sort(a, lo, hi)
if (lo < hi)
    int mid;
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)
```

What is the time for split?

$$\Theta(\lg n)$$

How many times does it execute?

$$\Theta(n)$$
The performance of `HeapSorter::sort()`

```cpp
sort(a, lo, hi)
if (lo < hi)
    int mid;
split(a, lo, mid, hi)
sort(a, lo, mid - 1)
```

What is the time for split?

\[ \Theta(\lg n) \]

How many times does it execute?

\[ \Theta(n) \]

Time for `sort()`
The performance of HeapSorter::sort()

sort(a, lo, hi)
if (lo < hi)
    int mid;
    split(a, lo, mid, hi)
    sort(a, lo, mid - 1)

What is the time for split?
\( \Theta(\lg n) \)

How many times does it execute?
\( \Theta(n) \)

Time for sort()
\( \Theta(n\lg n) \)
The performance of HeapSorter

Time to build initial heap + time to sort
The performance of HeapSorter

Time to build initial heap + time to sort

\[ \Theta(n \lg n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n) \]
The performance of HeapSorter

Time to build initial heap + time to sort

\[ \Theta(n \lg n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n) \]

Assuming tighter bound to build initial heap
The performance of HeapSorter

Time to build initial heap + time to sort

\[ \Theta(n \lg n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n) \]

Assuming tighter bound to build initial heap

\[ \Theta(n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n) \]