Chapter Objectives

- To learn how to use the standard sorting functions in `algorithm.h`
- To learn how to implement the following sorting algorithms:
  - selection sort
  - bubble sort
  - insertion sort
  - Shell sort
  - merge sort
  - heapsort
  - quicksort
- To understand the difference in performance of these algorithms, and which to use for small, medium, and large arrays
Introduction

- Sorting entails arranging data in decreasing (or nonincreasing) or increasing (or nondecreasing) order
- Familiarity with sorting algorithms is an important programming skill
- The study of sorting algorithms provides insight into
  - problem solving techniques such as divide and conquer
  - the analysis and comparison of algorithms which perform the same task
The Standard C++ Library (in `algorithm.h`) provides two sorting functions:

Each function uses a pair of random-access iterators (`typename RI`):

```cpp
template<typename RI>
void sort(RI first, RI last);

template<typename RI, typename Compare>
void sort(RI first, RI last, Compare comp);
```
Each function uses a pair of random access iterators 
\( \texttt{typename RI} \)

The first iterator references the first element of the sequence to be sorted

The second iterator references one past the last element in the sequence
Because these functions require random-access iterators, they can be used only with vectors, deques, and ordinary C pointers (for sorting arrays).

The list container supports only bidirectional iterators, so it provides its own sort member function.
There are also two functions named `stable_sort`, which are similar to the sort functions.

- The primary difference is that elements that are equal may not retain their relative ordering when `sort` is used, but they will retain their relative ordering when `stable_sort` is used.

- In other words, if there are two occurrences of an item, the one that is first in the unsorted array is guaranteed to be first in the sorted array only if `stable_sort` is used.

- The `sort` function is slightly faster than `stable_sort`, so it should be used whenever the relative ordering of equal elements is unimportant.
The actual type of random-access iterator passed to function `sort` depends on whether we are sorting an array, vector, or deque.

The compiler attempts to match the actual types of the arguments to the template parameters.

If we use the second version of function `sort`, we must also pass an object that implements a comparison function.

```cpp
bool myfunction (int i, int j) { return (i < j); }

struct myclass
{
  bool operator() (int i, int j) { return (i < j); }
}

myobject;
```
If array items stores a collection of 16 integers, the statement
\[
\text{sort}(\text{items}, \text{items} + 16);
\]
sorts the array.

The statement
\[
\text{sort}(\text{items}, \text{items} + 8);
\]
sorts only the first half of the array, leaving the rest untouched.

The function call
\[
\text{sort}(\text{items}, \text{items} + 16, \text{greater}<\text{int}>());
\]
sorts the array in descending order using function
\[
\text{greater}<\text{int}>, \text{which implements the comparison operator} \ > \text{for integers (defined in library}\ 
\text{functional})
The following statements sort the elements of `vector v` and `deque d`:

```cpp
sort(v.begin(), v.end());
sort(d.begin(), d.end());
```

The following statement shows how to sort the elements in `vector v` in descending order by using the comparison function `greater<int>()`:

```cpp
sort(v.begin(), v.end(), greater<int>());
```
In Section 9.2, we wrote the following `Compare_Person` function class:

```cpp
struct Compare_Person {
    bool operator()(const Person& p1, const Person& p2) {
        return (p1.family_name < p2.family_name)
        || (p1.family_name == p2.family_name)
        && (p1.given_name < p2.given_name);
    }
}
```

If `people_list` is a vector of `Person` objects, the statement

```cpp
sort(people_list.begin(), people_list.end(), Compare_Person());
```

sorts the elements in `people_list` in ascending order based on their names.

The client program must include header file "People.h", which defines class `Person` and struct `Compare_Person"
Selection Sort

- Selection sort is relatively easy to understand.
- It sorts an array by making several passes through the array, selecting a next smallest item in the array each time and placing it where it belongs in the array.
- For simplicity, we illustrate all the sorting algorithms using an array of integer values.
Trace of Selection Sort

\[ n = \text{number of elements in the array} \]

1. **for** \( \text{fill} = 0 \) to \( n - 2 \) **do**
2. Set \( \text{pos_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)
3. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>35</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>( n )</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \text{fill} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{pos_min} )</td>
<td></td>
</tr>
</tbody>
</table>
Trace of Selection Sort (cont.)

n = number of elements in the array

1. for fill = 0 to n - 2 do
   2. Set pos_min to the subscript of a smallest item in the subarray starting at subscript fill
   3. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td></td>
</tr>
</tbody>
</table>
n = number of elements in the array

1. for fill = 0 to n - 2 do
   2. Set pos_min to the subscript of a smallest item in the subarray starting at subscript fill
   3. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort (cont.)

\[ n = \text{number of elements in the array} \]

1. for \( \text{fill} = 0 \) to \( n - 2 \) do
   2. Set \( \text{pos\_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)
   3. Exchange the item at \( \text{pos\_min} \) with the one at \( \text{fill} \)

\[ \begin{array}{c|c|c|c|c}
0 & 1 & 2 & 3 & 4 \\
\hline
20 & 65 & 30 & 60 & 35 \\
\end{array} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fill} )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( \text{pos_min} )</td>
<td>( 4 )</td>
</tr>
</tbody>
</table>
n = number of elements in the array

1. for fill = 0 to n - 2 do
2. Set pos_min to the subscript of a smallest item in the subarray starting at subscript fill
3. Exchange the item at pos_min with the one at fill
Trace of Selection Sort (cont.)

\[ n = \text{number of elements in the array} \]

1. for \( \text{fill} = 0 \) to \( n - 2 \) do

   2. Set \( \text{pos_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)

   3. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)

\begin{align*}
\text{fill} & \quad \text{pos_min} \\
0 & \quad 1 \\
20 & \quad 65 \\
1 & \quad 2 \\
30 & \quad 60 \\
3 & \quad 4 \\
60 & \quad 35 \\
4 & \quad \text{fill} \\
\end{align*}

\begin{tabular}{|c|c|}
\hline
\text{n} & 5 \\
\hline
\text{fill} & 1 \\
\hline
\text{pos_min} & 2 \\
\hline
\end{tabular}
Trace of Selection Sort (cont.)

\[ n = \text{number of elements in the array} \]

1. for \( \text{fill} = 0 \) to \( n - 2 \) do
2. Set \( \text{pos_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)
3. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)

\[
\begin{array}{c|c|c|c|c|c}
0 & 1 & 2 & 3 & 4 \\
20 & 30 & 65 & 60 & 35 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>fill</th>
<th>pos_min</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Trace of Selection Sort (cont.)

\[ n = \text{number of elements in the array} \]

1. **for** \( \text{fill} = 0 \) to \( n - 2 \) do

2. Set \( \text{pos_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)

3. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)

---

<table>
<thead>
<tr>
<th>( n )</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fill} )</td>
<td>2</td>
</tr>
<tr>
<td>( \text{pos_min} )</td>
<td>2</td>
</tr>
</tbody>
</table>
n = number of elements in the array

1. for fill = 0 to n - 2 do
   2. Set pos_min to the subscript of a smallest item in the subarray starting at subscript fill
   3. Exchange the item at pos_min with the one at fill

<p>| | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>65</td>
<td>60</td>
<td>35</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5</td>
</tr>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_min</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort (cont.)

$n = \text{number of elements in the array}$

1. for $fill = 0$ to $n - 2$ do
2. Set $pos_{\text{min}}$ to the subscript of a smallest item in the subarray starting at subscript $fill$
3. Exchange the item at $pos_{\text{min}}$ with the one at $fill$

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_{\text{min}}</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort (cont.)

\[ n = \text{number of elements in the array} \]

1. for \( \text{fill} = 0 \) to \( n - 2 \) do

2. Set \( \text{pos\_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)

3. Exchange the item at \( \text{pos\_min} \) with the one at \( \text{fill} \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fill} )</td>
<td>3</td>
</tr>
<tr>
<td>( \text{pos_min} )</td>
<td>4</td>
</tr>
</tbody>
</table>
n = number of elements in the array

1. for fill = 0 to n - 2 do

2. Set pos_min to the subscript of a smallest item in the subarray starting at subscript fill

3. Exchange the item at pos_min with the one at fill

```
0 1 2 3 4
20 30 35 60 65
```

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>3</td>
</tr>
<tr>
<td>pos_min</td>
<td>3</td>
</tr>
</tbody>
</table>

Trace of Selection Sort (cont.)

\( n = \text{number of elements in the array} \)

1. for \( \text{fill} = 0 \) to \( n - 2 \) do
2. Set \( \text{pos_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)
3. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)

```
0 1 2 3 4
20 30 35 60 65
```

<table>
<thead>
<tr>
<th>( n )</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fill} )</td>
<td>3</td>
</tr>
<tr>
<td>( \text{pos_min} )</td>
<td>3</td>
</tr>
</tbody>
</table>
Trace of Selection Sort (cont.)

\[ n = \text{number of elements in the array} \]

1. for \( \text{fill} = 0 \) to \( n - 2 \) do
2. Set \( \text{pos_min} \) to the subscript of a smallest item in the subarray starting at subscript \( \text{fill} \)
3. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>30</td>
<td>35</td>
<td>60</td>
<td>65</td>
</tr>
</tbody>
</table>

| |  
|---|---|
| \( n \) | 5  
| \( \text{fill} \) | 3  
| \( \text{pos_min} \) | 3  |
Trace of Selection Sort Refinement

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td></td>
</tr>
<tr>
<td>pos_min</td>
<td></td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td></td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
</tbody>
</table>

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do

2. Initialize pos_min to fill

3. for next = fill + 1 to n - 1 do

4. if the item at next is less than the item at pos_min

5. Reset pos_min to next

6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>0</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>
```
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fill</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pos_min</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
1. for fill = 0 to n - 2 do
   2. Initialize pos_min to fill
   3. for next = fill + 1 to n - 1 do
      4. if the item at next is less than the item at pos_min
         5. Reset pos_min to next
      6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
   4. if the item at next is less than the item at pos_min
      Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
   5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>2</td>
</tr>
</tbody>
</table>

```
0 1 2 3 4
35 65 30 60 20
```
Trace of Selection Sort Refinement (cont.)

1. for \( \text{fill} = 0 \) to \( n - 2 \) do
2. Initialize \( \text{pos_min} \) to \( \text{fill} \)
3. for \( \text{next} = \text{fill} + 1 \) to \( n - 1 \) do
4. if the item at \( \text{next} \) is less than the item at \( \text{pos_min} \)
5. Reset \( \text{pos_min} \) to \( \text{next} \)
6. Exchange the item at \( \text{pos_min} \) with the one at \( \text{fill} \)
Trace of Selection Sort Refinement
(cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
   Reset pos_min to next
5. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>3</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement
(cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>35</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>
```
1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
   4. if the item at next is less than the item at pos_min
      5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>

fill pos_min next
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>4</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
   Reset pos_min to next
5. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>0</td>
</tr>
<tr>
<td>pos_min</td>
<td>4</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n – 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n – 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
   2. Initialize pos_min to fill
   3. for next = fill + 1 to n - 1 do
   4. if the item at next is less than the item at pos_min
      Reset pos_min to next
   5. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>1</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement
(cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>1</td>
</tr>
<tr>
<td>next</td>
<td>2</td>
</tr>
</tbody>
</table>

1. \textbf{for} fill = 0 \textbf{to} n - 2 \textbf{do}

2. Initialize pos_min to fill

3. \textbf{for} next = fill + 1 \textbf{to} n - 1 \textbf{do}

4. \textbf{if} the item at next is less than the item at pos_min

5. Reset pos_min to next

6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>1</td>
</tr>
<tr>
<td>next</td>
<td>2</td>
</tr>
</tbody>
</table>

1. **for** fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. **for** next = fill + 1 to n - 1 do
   4. if the item at next is less than the item at pos_min
      Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement
(cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>2</td>
</tr>
</tbody>
</table>

```
fill   pos_min   next
0 1 2 3 4
20 65 30 60 35
```
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>3</td>
</tr>
</tbody>
</table>

1. for fill = 0 to n – 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n – 1 do
   ▶ 4. if the item at next is less than the item at pos_min
      5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
   ➤ 3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
### Trace of Selection Sort Refinement
(cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>1</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

1. **for** fill = 0 to n – 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n – 1 do
4.    if the item at next is less than the item at pos_min
5.    Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>65</td>
<td>30</td>
<td>60</td>
<td>35</td>
</tr>
</tbody>
</table>

- **fill**
- **pos_min**
- **next**
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_min</td>
<td>2</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

1. **for** fill = 0 to n - 2 do

2. Initialize pos_min to fill

3. **for** next = fill + 1 to n - 1 do

4. **if** the item at next is less than the item at pos_min

5. Reset pos_min to next

6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
   Reset pos_min to next
5. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. \textbf{for} fill = 0 \textbf{to} n - 2 \textbf{do}
2. \hspace{1em} \textbf{Initialize} pos_min \textbf{to} fill
3. \hspace{1em} \textbf{for} next = fill + 1 \textbf{to} n - 1 \textbf{do}
4. \hspace{2em} \textbf{if} the item at next \textbf{is less than} the item at pos_min
5. \hspace{3em} \textbf{Reset} pos_min \textbf{to} next
6. \hspace{1em} \textbf{Exchange} the item at pos_min \textbf{with the one at} fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_min</td>
<td>3</td>
</tr>
<tr>
<td>next</td>
<td>3</td>
</tr>
</tbody>
</table>

0 1 2 3 4

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
 fill & pos_min & next & 0 & 1 & 2 & 3 & 4 \\
\hline
 20 & 30 & 65 & 60 & 35 & & & \\
\hline
\end{array}
\]
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_min</td>
<td>3</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
   4. if the item at next is less than the item at pos_min
   5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. **for** fill = 0 **to** n - 2 **do**
2. **Initialize** pos_min **to** fill
3. **for** next = fill + 1 **to** n - 1 **do**
   4. **if** the item at next **is less than the** item at pos_min
      5. **Reset** pos_min **to** next
6. **Exchange** the item at pos_min **with the** one at fill
1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>2</td>
</tr>
<tr>
<td>pos_min</td>
<td>4</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n – 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n – 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do

2. Initialize pos_min to fill

3. for next = fill + 1 to n - 1 do

4. if the item at next is less than the item at pos_min

5. Reset pos_min to next

6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>3</td>
</tr>
<tr>
<td>pos_min</td>
<td>3</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

0 1 2 3 4

20 30 35 60 65
### Trace of Selection Sort Refinement (cont.)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>n</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>fill</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>pos_min</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>next</strong></td>
<td>4</td>
</tr>
</tbody>
</table>

1. **for** fill = 0 to n - 2 do  
2. Initialize pos_min to fill  
3. **for** next = fill + 1 to n - 1 do  
4. if the item at next is less than the item at pos_min  
5. Reset pos_min to next  
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n – 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n – 1 do
   4. if the item at next is less than the item at pos_min
      Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>3</td>
</tr>
<tr>
<td>pos_min</td>
<td>3</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>
Trace of Selection Sort Refinement (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill
Trace of Selection Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>3</td>
</tr>
<tr>
<td>pos_min</td>
<td>3</td>
</tr>
<tr>
<td>next</td>
<td>4</td>
</tr>
</tbody>
</table>

1. for fill = 0 to n – 2 do
   2. Initialize pos_min to fill
   3. for next = fill + 1 to n – 1 do
      4. if the item at next is less than the item at pos_min
         Reset pos_min to next
   5. Exchange the item at pos_min with the one at fill
Analysis of Selection Sort

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
   Reset pos_min to next
5. Exchange the item at pos_min with the one at fill

This loop is performed n-1 times
Analysis of Selection Sort (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
   Reset pos_min to next
5. Exchange the item at pos_min with the one at fill

There are n-1 exchanges
1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

This comparison is performed \((n - 1 - \text{fill})\) times for each value of \text{fill} and can be represented by the following series:
\((n-1) + (n-2) + ... + 3 + 2 + 1\)
Analysis of Selection Sort (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

The series \( (n-1) + (n-2) + \ldots + 3 + 2 + 1 \) is a well-known series and can be written as

\[
\frac{n \times (n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2}
\]
Analysis of Selection Sort (cont.)

1. for fill = 0 to n - 2 do
2. Initialize pos_min to fill
3. for next = fill + 1 to n - 1 do
4. if the item at next is less than the item at pos_min
5. Reset pos_min to next
6. Exchange the item at pos_min with the one at fill

For very large n we can ignore all but the significant term in the expression, so the number of
• comparisons is $O(n^2)$
• exchanges is $O(n)$

An $O(n^2)$ sort is called a quadratic sort
Bubble Sort

- Also a quadratic sort
- Compares adjacent array elements and exchanges their values if they are out of order
- Smaller values *bubble* up to the top of the array and larger values sink to the bottom; hence the name
Trace of Bubble Sort

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

```
[0] 60
[1] 42
[2] 75
[3] 83
[4] 27
```
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>2</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

1. **do**
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

At the end of pass 1, the last item (index [4]) is guaranteed to be in its correct position. There is no need to test it again in the next pass.
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

1. **do**
2. **for** each pair of adjacent array elements
3. **if** the values in a pair are out of order
4. Exchange the values
5. **while** the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>0</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>Pass</th>
<th>Exchanges Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

<table>
<thead>
<tr>
<th>pass</th>
<th>exchanges made</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

[0] 42
[1] 27
[2] 60
[3] 75
[4] 83
1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted
Trace of Bubble Sort (cont.)

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

<table>
<thead>
<tr>
<th>pass</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

[0] 27
[1] 42
[2] 60
[3] 75
[4] 83
## Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

| [0] 27 |
| [1] 42 |
| [2] 60 |
| [3] 75 |
| [4] 83 |

Where \( n \) is the length of the array, after the completion of \( n - 1 \) passes (4, in this example) the array is sorted.
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

Sometimes an array will be sorted before $n - 1$ passes. This can be detected if there are no exchanges made during a pass through the array.
Trace of Bubble Sort (cont.)

<table>
<thead>
<tr>
<th>pass</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>exchanges made</td>
<td>1</td>
</tr>
</tbody>
</table>

1. do
2. for each pair of adjacent array elements
3. if the values in a pair are out of order
4. Exchange the values
5. while the array is not sorted

The algorithm can be modified to detect exchanges (next)
Refinement of Bubble Sort Algorithm (Steps 2-4)

2.1 Initialize exchanges to false

2.2 for each pair of adjacent array elements

3. if the values in a pair are out of order

4.1 Exchange the values

4.2 Set exchanges to true
Analysis of Bubble Sort

- The number of comparisons and exchanges is represented by
  \[(n - 1) + (n - 2) + ... + 3 + 2 + 1\]

- Worst case:
  - Number of comparisons is \(O(n^2)\)
  - Number of exchanges is \(O(n^2)\)

- Compared to selection sort with its \(O(n^2)\) comparisons and \(O(n)\) exchanges, bubble sort usually performs worse.

- If the array is sorted early, the later comparisons and exchanges are not performed and performance is improved.
Analysis of Bubble Sort (cont.)

- The best case occurs when the array is sorted already
  - one pass is required (O(n) comparisons)
  - no exchanges are required (O(1) exchanges)
- Bubble sort works well on arrays nearly sorted and worst on inverted arrays (elements are in reverse sorted order)
Insertion Sort

Section 10.4
Insertion Sort

- Another quadratic sort, *insertion* sort, is based on the technique used by card players to arrange a hand of cards.
  - The player keeps the cards that have been picked up so far in sorted order.
  - When the player picks up a new card, the player makes room for the new card and then inserts it in its proper place.

![Image of playing cards showing sorted and unsorted hands.](image-url)
Insertion Sort (cont.)

- Start with a sorted subarray, consisting of only the first element
- Insert the second element either in front of or behind the first element, producing a sorted subarray of size 2
- Insert the third element in front of, between, or behind the first two elements
- Continue until all elements have been inserted
Trace of Insertion Sort

1. for each array element from the second (next_pos = 1) to the last
2. Insert the element at next_pos where it belongs in the array, increasing the length of the sorted subarray by 1 element

To adapt the insertion algorithm to an array that is filled with data, we start with a sorted subarray consisting of only the first element
Trace of Insertion Sort (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. Insert the element at next_pos where it belongs in the array, increasing the length of the sorted subarray by 1 element

<table>
<thead>
<tr>
<th>next_pos</th>
<th>1</th>
</tr>
</thead>
</table>

```
[0] 30
[1] 25
[2] 15
[3] 20
[4] 28
```
1. for each array element from the second (\(next\_pos = 1\)) to the last
2. Insert the element at \(next\_pos\) where it belongs in the array, increasing the length of the sorted subarray by 1 element

\[
\begin{array}{c|c|c}
\text{next_pos} & 1 \\
\hline
[0] & 25 & \text{next_pos} \\
[1] & 30 & \text{next_pos} \\
[2] & 15 & \\
[3] & 20 & \\
[4] & 28 & \\
\end{array}
\]
Trace of Insertion Sort (cont.)

1. for each array element from the second (\texttt{next\_pos} = 1) to the last

2. Insert the element at \texttt{next\_pos} where it belongs in the array, increasing the length of the sorted subarray by 1 element
Trace of Insertion Sort (cont.)

1. For each array element from the second (next_pos = 1) to the last

2. Insert the element at next_pos where it belongs in the array, increasing the length of the sorted subarray by 1 element
1. **for** each array element from the second (next_pos = 1) to the last

2. Insert the element at `next_pos` where it belongs in the array, increasing the length of the sorted subarray by 1 element
1. for each array element from the second (next_pos = 1) to the last
2. Insert the element at next_pos where it belongs in the array, increasing the length of the sorted subarray by 1 element
Trace of Insertion Sort (cont.)

1. for each array element from the second (next_pos = 1) to the last

2. Insert the element at next_pos where it belongs in the array, increasing the length of the sorted subarray by 1 element
Trace of Insertion Sort (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. Insert the element at next_pos where it belongs in the array, increasing the length of the sorted subarray by 1 element
Trace of Insertion Sort (cont.)

1. for each array element from the second 
   (next_pos = 1) to the last 
2. Insert the element at next_pos where it 
   belongs in the array, increasing the length of 
   the sorted subarray by 1 element
1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next_pos</td>
<td>1</td>
</tr>
<tr>
<td>next_val</td>
<td>25</td>
</tr>
</tbody>
</table>

1. **for** each array element from the second (\( \text{next\_pos} = 1 \)) to the last
2. \( \text{next\_pos} \) is the position of the element to insert
3. Save the value of the element to insert in \( \text{next\_val} \)
4. **while** \( \text{next\_pos} > 0 \) and the element \( \text{at} \) \( \text{next\_pos} - 1 > \text{next\_val} \)
5. Shift the element \( \text{at} \) \( \text{next\_pos} - 1 \) to position \( \text{next\_pos} \)
6. Decrement \( \text{next\_pos} \) by 1
7. **Insert** \( \text{next\_val} \) \( \text{at} \) \( \text{next\_pos} \)
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos

<table>
<thead>
<tr>
<th>next_pos</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>25</td>
</tr>
</tbody>
</table>
Trace of Insertion Sort Refinement (cont.)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next_pos</td>
<td>1</td>
</tr>
<tr>
<td>next_val</td>
<td>25</td>
</tr>
</tbody>
</table>

1. **for** each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. **while** next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>25</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
### Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>25</td>
</tr>
</tbody>
</table>

1. **for** each array element from the second (next_pos = 1) to the last  
2. next_pos is the position of the element to insert  
3. Save the value of the element to insert in next_val  
4. **while** next_pos > 0 and the element at next_pos - 1 > next_val  
5. Shift the element at next_pos - 1 to position next_pos  
6. Decrement next_pos by 1  
7. Insert next_val at next_pos
1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos

<table>
<thead>
<tr>
<th>next_pos</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>25</td>
</tr>
</tbody>
</table>
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next_pos</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next_val</td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos

<table>
<thead>
<tr>
<th>next_pos</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>15</td>
</tr>
</tbody>
</table>
Trace of Insertion Sort Refinement (cont.)

| next_pos | 2 |
| next_val | 15 |

1. **for** each array element from the second (next_pos = 1) to the last

2. **next_pos** is the position of the element to insert

3. Save the value of the element to insert in **next_val**

4. **while** next_pos > 0 and the element at next_pos - 1 > next_val

5. **Shift** the element at next_pos - 1 to position next_pos

6. **Decrement** next_pos by 1

7. **Insert** next_val at next_pos

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next_pos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>15</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
   5. Shift the element at next_pos - 1 to position next_pos
   6. Decrement next_pos by 1
5. Insert next_val at next_pos

<table>
<thead>
<tr>
<th>loop position</th>
<th>next_pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 25</td>
<td></td>
</tr>
<tr>
<td>[1] 25</td>
<td></td>
</tr>
<tr>
<td>[2] 30</td>
<td></td>
</tr>
<tr>
<td>[3] 20</td>
<td></td>
</tr>
<tr>
<td>[4] 28</td>
<td></td>
</tr>
</tbody>
</table>
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>15</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>15</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos

loop position next_pos

[0] 15
[1] 25
[2] 30
[3] 20
[4] 28
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>20</td>
</tr>
</tbody>
</table>

1. **for** each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. **while** next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. for each array element from the second (\( \text{next_pos} = 1 \)) to the last
2. \( \text{next_pos} \) is the position of the element to insert
3. Save the value of the element to insert in \( \text{next_val} \)
4. while \( \text{next_pos} > 0 \) and the element at \( \text{next_pos} - 1 > \text{next_val} \)
5. Shift the element at \( \text{next_pos} - 1 \) to position \( \text{next_pos} \)
6. Decrement \( \text{next_pos} \) by 1
7. Insert \( \text{next_val} \) at \( \text{next_pos} \)
Trace of Insertion Sort Refinement
(cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
   5. Shift the element at next_pos - 1 to position next_pos
5. Decrement next_pos by 1
6. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. **for** each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. **while** next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>20</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. for each array element from the second
   (next_pos = 1) to the last
2. next_pos is the position of the element
to insert
3. Save the value of the element to insert in
next_val
4. while next_pos > 0 and the element
   at next_pos - 1 > next_val
5. Shift the element at next_pos - 1
to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement
(cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>20</td>
</tr>
</tbody>
</table>

1. **for** each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. **while** next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos – 1 > next_val

5. Shift the element at next_pos – 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>28</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

1. for each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. while next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>28</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
   5. Shift the element at next_pos - 1 to position next_pos
   6. Decrement next_pos by 1
7. Insert next_val at next_pos
1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>28</td>
</tr>
</tbody>
</table>

1. **for** each array element from the second (next_pos = 1) to the last

2. next_pos is the position of the element to insert

3. Save the value of the element to insert in next_val

4. **while** next_pos > 0 and the element at next_pos - 1 > next_val

5. Shift the element at next_pos - 1 to position next_pos

6. Decrement next_pos by 1

7. Insert next_val at next_pos
Trace of Insertion Sort Refinement (cont.)

<table>
<thead>
<tr>
<th>next_pos</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_val</td>
<td>28</td>
</tr>
</tbody>
</table>

1. for each array element from the second (next_pos = 1) to the last
2. next_pos is the position of the element to insert
3. Save the value of the element to insert in next_val
4. while next_pos > 0 and the element at next_pos - 1 > next_val
5. Shift the element at next_pos - 1 to position next_pos
6. Decrement next_pos by 1
7. Insert next_val at next_pos
Analysis of Insertion Sort

- The insertion step is performed \( n - 1 \) times.
- In the worst case, all elements in the sorted subarray are compared to `next_val` for each insertion.
- The maximum number of comparisons then will be:
  \[
  1 + 2 + 3 + ... + (n - 2) + (n - 1)
  \]
- Which is \( O(n^2) \).
Analysis of Insertion Sort (cont.)

- In the best case (when the array is sorted already), only one comparison is required for each insertion.
- In the best case, the number of comparisons is $O(n)$.
- The number of shifts performed during an insertion is one less than the number of comparisons.
- Or, when the new value is the smallest so far, it is the same as the number of comparisons.
- A shift in an insertion sort requires movement of only 1 item, while an exchange in a bubble or selection sort involves a temporary item and the movement of three items.
  - A C++ array of objects contains the actual objects, and it is these actual objects that are changed.
## Comparison of Quadratic Sorts

<table>
<thead>
<tr>
<th></th>
<th>Number of Comparisons</th>
<th>Number of Exchanges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Worst</td>
</tr>
<tr>
<td>Selection sort</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>
Comparison of Quadratic Sorts (cont.)

Comparison of growth rates

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>1,024</td>
<td>160</td>
</tr>
<tr>
<td>64</td>
<td>4,096</td>
<td>384</td>
</tr>
<tr>
<td>128</td>
<td>16,384</td>
<td>896</td>
</tr>
<tr>
<td>256</td>
<td>65,536</td>
<td>2,048</td>
</tr>
<tr>
<td>512</td>
<td>262,144</td>
<td>4,608</td>
</tr>
</tbody>
</table>
Comparison of Quadratic Sorts (cont.)

- Insertion sort
  - gives the best performance for most arrays
  - takes advantage of any partial sorting in the array and uses less costly shifts

- Bubble sort generally gives the worst performance—unless the array is nearly sorted
  - Big-O analysis ignores constants and overhead

- None of the quadratic search algorithms are particularly good for large arrays ($n > 100$)

- The best sorting algorithms provide $n \log n$ average case performance
Comparison of Quadratic Sorts (cont.)

- All quadratic sorts require storage for the array being sorted
- However, the array is sorted in place
- While there are also storage requirements for variables, for large $n$, the size of the array dominates and extra space usage is $O(1)$
Comparisons versus Exchanges

- In C++, an exchange requires a swap of two objects using a third object as an intermediary.
- A comparison requires an evaluation of the `<` operator, or a call to a comparison function.
- The cost of a comparison depends on its complexity, but is generally more costly than an exchange.
- The cost of an exchange is proportional to the size of the objects being exchanged and, therefore, may be more costly than a comparison for large objects.
Section 10.6

Shell Sort: A Better Insertion Sort
Shell Sort: A Better Insertion Sort

- A Shell sort is a type of insertion sort, but with $O(n^{3/2})$ or better performance than the $O(n^2)$ sorts.
- It is named after its discoverer, Donald Shell.
- A Shell sort can be thought of as a divide-and-conquer approach to insertion sort.
- Instead of sorting the entire array, Shell sort sorts many smaller subarrays using insertion sort before sorting the entire array.
### Trace of Shell Sort

| gap value | 7 |

<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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>35</td>
<td>80</td>
<td>75</td>
<td>60</td>
<td>90</td>
<td>70</td>
<td>75</td>
<td>55</td>
<td>90</td>
<td>85</td>
<td>34</td>
<td>45</td>
<td>62</td>
<td>57</td>
<td>65</td>
</tr>
</tbody>
</table>
Trace of Shell Sort (cont.)

| gap value | 7 |

subarray 1


40 35 80 75 60 90 70 75 55 90 85 34 45 62 57 65
Trace of Shell Sort (cont.)

| gap value | 7 |

Subarray 2
Trace of Shell Sort (cont.)

- gap value: 7

- subarray 3
Trace of Shell Sort (cont.)

Gap value: 7

Subarray 4
Trace of Shell Sort (cont.)

| gap value | 7 |

subarray 5
Trace of Shell Sort (cont.)

Gap value: 7

Subarray 6
Trace of Shell Sort (cont.)

| gap value | 7 |

![Diagram of Shell Sort with gap value 7 and subarray 7 highlighted]
Trace of Shell Sort (cont.)

Gap value: 7

Sort subarray 1

Subarray 1:

[0] 40 35 80 75 60 90 70 75 55 90 85 34 45 62 57 65
Trace of Shell Sort (cont.)

gap value 7

Sort subarray 2

subarray 2
Trace of Shell Sort (cont.)

Gap value 7

Sort subarray 3

Subarray 3
Trace of Shell Sort (cont.)

| gap value | 7 |

Sort subarray 4

Subarray 4: [40, 35, 80, 75, 60, 90, 70, 75, 55, 90, 85, 34, 45, 62, 57, 65]
Trace of Shell Sort (cont.)

- **Gap Value**: 7

Diagram showing the array with a highlighted subarray 5, and an instruction to sort subarray 5.
Trace of Shell Sort (cont.)

- **gap value**: 7

**Sort subarray 5**

```
40   35   80   75   34   90   70   75   55   90   85   60   45   62   57   65
```

**subarray 5**
Trace of Shell Sort (cont.)

gap value 7

Sort subarray 6

subarray 6
Trace of Shell Sort (cont.)

Gap value: 7

Sort subarray 6

Subarray 6
Trace of Shell Sort (cont.)

| gap value | 7 |

Sort subarray 7

subarray 7
Trace of Shell Sort (cont.)

| gap value | 7 |

Sort subarray 7

subarray 7
Trace of Shell Sort (cont.)

- Gap value: 7

Subarray 1
Trace of Shell Sort (cont.)

| gap value | 7 |

Sort subarray 1

subarray 1
Trace of Shell Sort (cont.)

| gap value | 7 |

Sort subarray 2

subarray 2
Trace of Shell Sort (cont.)

| gap value | 7 |

Sort on smaller gap value next

<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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>35</td>
<td>80</td>
<td>75</td>
<td>34</td>
<td>45</td>
<td>62</td>
<td>57</td>
<td>55</td>
<td>90</td>
<td>85</td>
<td>60</td>
<td>90</td>
<td>70</td>
<td>75</td>
<td>65</td>
</tr>
</tbody>
</table>
### Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>3</th>
</tr>
</thead>
</table>

<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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>35</td>
<td>80</td>
<td>75</td>
<td>34</td>
<td>45</td>
<td>62</td>
<td>57</td>
<td>55</td>
<td>90</td>
<td>85</td>
<td>60</td>
<td>90</td>
<td>70</td>
<td>75</td>
<td>65</td>
</tr>
</tbody>
</table>
Trace of Shell Sort (cont.)

Gap value: 3

Sort subarray 1

Subarray 1 is the set of elements from index 0 to index 3.
Trace of Shell Sort (cont.)

gap value 3

Sort subarray 2
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 2

subarray 2
Trace of Shell Sort (cont.)

gap value 3

Sort subarray 3

subarray 3
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 3

subarray 3
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 1

subarray 1

<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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>34</td>
<td>45</td>
<td>75</td>
<td>35</td>
<td>80</td>
<td>62</td>
<td>57</td>
<td>55</td>
<td>90</td>
<td>85</td>
<td>60</td>
<td>90</td>
<td>70</td>
<td>75</td>
<td>65</td>
</tr>
</tbody>
</table>
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 1

Subarray 1
Trace of Shell Sort (cont.)

- **gap value**: 3

Sort subarray 2

```
40  34  45  62  35  80  75  57  55  90  85  60  90  70  75  65
```

subarray 2
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>3</th>
</tr>
</thead>
</table>

Sort subarray 3

subarray 3
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>3</th>
</tr>
</thead>
</table>

Sort subarray 3

subarray 3
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 1

subarray 1
Trace of Shell Sort (cont.)

gap value | 3

Sort subarray 2

subarray 2
Trace of Shell Sort (cont.)

Gap value: 3

Sort subarray 3

Subarray 3
Trace of Shell Sort (cont.)

- **gap value**: 3
- **Sort subarray 3**

The diagram illustrates the sorting process with a gap value of 3, focusing on subarray 3.
Trace of Shell Sort (cont.)

- **Gap Value**: 3

- **Sort Subarray 1**

```
40  34  45  62  35  55  75  57  60  90  85  80  90  70  75  65
```

**Subarray 1**
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 2

subarray 2
Trace of Shell Sort (cont.)

gap value 3

Sort subarray 2

subarray 2
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 3

Subarray 3
Trace of Shell Sort (cont.)

- Gap value: 3

Sort subarray 3

Subarray 3
Trace of Shell Sort (cont.)

| gap value | 3 |

Sort subarray 1

subarray 1
Trace of Shell Sort (cont.)

**gap value**

| 3 |

Sort subarray 1

```
```

- 40 34 45 62 35 55 75 57 60 65 70 75 90 85 80 90

subarray 1
Trace of Shell Sort (cont.)

gap value 3

Sort subarray 1

subarray 1
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>3</th>
</tr>
</thead>
</table>

Sort on gap value of 1  
(a regular insertion sort)

<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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>34</td>
<td>45</td>
<td>62</td>
<td>35</td>
<td>55</td>
<td>65</td>
<td>57</td>
<td>60</td>
<td>75</td>
<td>70</td>
<td>75</td>
<td>90</td>
<td>85</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>1</th>
</tr>
</thead>
</table>

Sort on gap value of 1 (a regular insertion sort)


40 34 45 62 35 55 65 57 60 75 70 75 90 85 80 90
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

gap value | 1
---|---

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>1</th>
</tr>
</thead>
</table>

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

gap value 1

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

- **gap value**: 1

  *Sort on gap value of 1 (a regular insertion sort)*
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

gap value 1

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>1</th>
</tr>
</thead>
</table>

Sort on gap value of 1
(a regular insertion sort)

<table>
<thead>
<tr>
<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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>55</td>
<td>57</td>
<td>60</td>
<td>62</td>
<td>65</td>
<td>75</td>
<td>70</td>
<td>75</td>
<td>90</td>
<td>85</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)

34 35 40 45 55 57 60 62 65 75 70 75 90 85 80 90
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)

<table>
<thead>
<tr>
<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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>55</td>
<td>57</td>
<td>60</td>
<td>62</td>
<td>65</td>
<td>70</td>
<td>75</td>
<td>75</td>
<td>90</td>
<td>85</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)
## Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>1</th>
</tr>
</thead>
</table>

Sort on gap value of 1 (a regular insertion sort)

![Shell Sort Trace](image)
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>1</th>
</tr>
</thead>
</table>

Sort on gap value of 1
(a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)

34  35  40  45  55  57  60  62  65  70  75  75  85  90  80  90
Trace of Shell Sort (cont.)

<table>
<thead>
<tr>
<th>gap value</th>
<th>1</th>
</tr>
</thead>
</table>

Sort on gap value of 1 (a regular insertion sort)
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1
(a regular insertion sort)


34 35 40 45 55 57 60 62 65 70 75 75 80 85 90 90
Trace of Shell Sort (cont.)

| gap value | 1 |

Sort on gap value of 1 (a regular insertion sort)
Shell Sort Algorithm

1. Set the initial value of gap to $\frac{n}{2}$
2. while $\text{gap} > 0$
3. for each array element from position gap to the last element
4. Insert this element where it belongs in its subarray.
5. if gap is 2, set it to 1
6. else gap = gap / 2. // chosen by experimentation
Shell Sort Algorithm (cont.)

Refinement of Step 4, the Insertion Step

4.1 \textit{next\_pos} is an iterator to the element to insert

4.2 Save the value of the element to insert in \textit{next\_val}

4.3 while \textit{next\_pos} > \textit{first} + \textit{gap} and the element at \textit{next\_pos} - \textit{gap} > \textit{next\_val}

4.4 Shift the element at \textit{next\_pos} - \textit{gap} to position \textit{next\_pos}

4.5 Decrement \textit{next\_pos} by \textit{gap}

4.6 Insert \textit{next\_val} at \textit{next\_pos}
Analysis of Shell Sort

- Because the behavior of insertion sort is closer to $O(n)$ than $O(n^2)$ when an array is nearly sorted, presorting speeds up later sorting.
- This is critical when sorting large arrays where the $O(n^2)$ performance becomes significant.
A general analysis of Shell sort is an open research problem in computer science.

Performance depends on how the decreasing sequence of values for \( \text{gap} \) is chosen.

If successive powers of 2 are used for \( \text{gap} \), performance is \( O(n^2) \).

If successive values for \( \text{gap} \) are based on Hibbard's sequence, \( 2^k - 1 \) (i.e. 31, 15, 7, 3, 1), it can be proven that the performance is \( O(n^{3/2}) \).

Other sequences give similar or better performance.
Our algorithm selects the initial value of $\text{gap}$ as $n/2$, divides by 2.2, and truncates the result.

Empirical studies show that the performance is $O(n^{5/4})$ or even $O(n^{7/6})$, but there is no theoretical basis for this result.
A merge is a common data processing operation performed on two sequences of data with the following characteristics:

- Both sequences are ordered by the same comparison operator (that is, both sequences are sorted)

The result of the merge operation is a third sequence containing all the data from the first two sequences.
Merge Algorithm

1. Access the first item from both sequences.
2. while not finished with either sequence
3. Compare the current items from the two sequences, copy the smaller current item to the output sequence, and access the next item from the input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.
Analysis of Merge

- For two input sequences each containing $n$ elements, each element needs to move from its input sequence to the output sequence.

- Merge time is $O(n)$.

- Space requirements:
  - The array cannot be merged in place.
  - Additional space usage is $O(n)$.
  - Vector!
Merge Sort

- We can modify merging to sort a single, unsorted array
  1. Split the array into two halves
  2. Sort the left half
  3. Sort the right half
  4. Merge the two

- This algorithm can be written with a recursive step
Algorithm for Merge Sort

1. if the table has more than one element
2. Store the first half of the table in `left_table`
3. Store the second half of the table in `right_table`
4. Recursively apply the merge sort algorithm to `left_table`
5. Recursively apply the merge sort algorithm to `right_table`
6. Call the merge function with `left_table` and `right_table` as the input sequences and the original table as the output sequence
Trace of Merge Sort

<table>
<thead>
<tr>
<th>50</th>
<th>60</th>
<th>45</th>
<th>30</th>
<th>90</th>
<th>20</th>
<th>80</th>
<th>15</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>50</th>
<th>60</th>
<th>45</th>
<th>30</th>
</tr>
</thead>
</table>

| 90 | 20 | 80 | 15 |
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)

50 60 45 30 90 20 80 15

50 60 45 30

90 20 80 15

45 30

45 30
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)

```
50  60  45  30  90  20  80  15

30  45  50  60
90  20  80  15
20  90
15  80
80  15
```
Trace of Merge Sort (cont.)
Trace of Merge Sort (cont.)

<table>
<thead>
<tr>
<th>15</th>
<th>20</th>
<th>30</th>
<th>45</th>
<th>50</th>
<th>60</th>
<th>80</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>45</td>
<td>50</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>80</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis of Merge Sort

- Each backward step requires a movement of $n$ elements from smaller-size arrays to larger arrays; the effort is $O(n)$
- The number of steps which require merging is $\log n$ because each recursive call splits the array in half
- The total effort to reconstruct the sorted array through merging is $O(n \log n)$
Analysis of Merge Sort (cont.)

- Going down through the recursion chain, sorting the left tables, a sequence of right tables of size \( \frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^k} \) is allocated.

- Since

\[
\frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1 = n - 1
\]

a total of \( n \) additional storage locations are required.
Heapsort

Section 10.8
Heapsort

- Merge sort time is $O(n \log n)$ but still requires, temporarily, $n$ extra storage locations
- Heapsort does not require any additional storage
- As its name implies, heapsort uses a heap to store the array
Heapsort Algorithm

- A max heap is a data structure that maintains the largest value at the top.
- Once we have a heap, we can remove one item at a time from the heap.
- The item removed is always the top element, and we will place it at the bottom of the heap.
- When we reheap, the larger of a node’s two children is moved up the heap, so the new heap will have the next largest item as its root.
- As we continue to remove items from the heap, the heap size shrinks as the number of the removed items increases.
- If we implement the heap using an array, each element removed will be placed at the end of the array, but in front of the elements that were removed earlier.
- After we remove the last element, the array will be sorted.
Trace of Heapsort
Trace of Heapsort (cont.)

```
89
/   \
76   \
|   |
37   76
     / \
    20  26
```

```
32
|   |
18   28
```

```
39
|   |
29   6
```

```
74
|   |
39   74
     / \n    66
```
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)

```
32
 /  \
26   29
 /    /  \
20   6   28
 /     /    /  \
37   39   66   74
 /     /    /    /  \
76   89
```
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)

Diagram of a heap with values 37, 39, 66, 74, 76, 89, 26, 28, 29, 18, and 20.
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Trace of Heapsort (cont.)
Revising the Heapsort Algorithm

- If we implement the heap as an array
  - each element removed will be placed at the end of the array, and
  - the heap part of the array decreases by one element
Algorithm for In-Place Heapsort

1. Build a heap by rearranging the elements in an unsorted array

2. while the heap is not empty

3. Remove the first item from the heap by swapping it with the last item in the heap and restoring the heap property
Algorithm to Build a Heap

- If we want to sort the sequence `table` bounded by the iterators `first` through `last`, we can consider the first item to be a heap of one item.

- We now consider the general case where the items in the sequence from `table[first]` through `table[first + n - 1]` form a heap; the items from `table[first + n]` through `table[last - 1]` are not in the heap.

- As each item is inserted, we must “reheap” to restore the heap property.
Refinement of Step 1 for In-Place Heapsort

1.1 while n is less than table.length

1.2 Increment n by 1. This inserts a new item into the heap

1.3 Restore the heap property
Analysis of Heapsort

- Because a heap of size $n$ is a complete binary tree, it has $\log n$ levels.
- Building a heap requires finding the correct location for an item in a heap with $\log n$ levels.
- Each insert (or remove) is $O(\log n)$.
- With $n$ items, building a heap is $O(n \log n)$.
- No extra storage is needed.
Quicksort

Section 10.9
Quicksort

- Developed in 1962
- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called partitioning)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
  - The pivot is placed between the two subarrays
- The process is repeated until the array is sorted
Trace of Quicksort
Arbitrarily select the first element as the pivot
Trace of Quicksort (cont.)

Swap the pivot with the element in the middle
Trace of Quicksort (cont.)

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right.
Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right.
Quicksort Example (cont.)

44 is now in its correct position

12 33 23 43 44 55 64 77 75
Trace of Quicksort (cont.)

Now apply quicksort recursively to the two subarrays.
Trace of Quicksort (cont.)

Pivot value = 12

12 33 23 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 12

12 33 23 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 33

12  33  23  43  44  55  64  77  75
Trace of Quicksort (cont.)

Pivot value = 33

12 23 33 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 33

12  23  33  43  44  55  64  77  75
Trace of Quicksort (cont.)

Pivot value = 33

Left and right subarrays have single values; they are sorted
Trace of Quicksort (cont.)

Pivot value = 33

Left and right subarrays have single values; they are sorted
Trace of Quicksort (cont.)

Pivot value = 55
Trace of Quicksort (cont.)

Pivot value = 64

12 23 33 43 44 55 64 77 75
Trace of Quicksort (cont.)

Pivot value = 77
Trace of Quicksort (cont.)

Pivot value = 77

12 23 33 43 44 55 64 75 77
Trace of Quicksort (cont.)

Pivot value = 77

12 23 33 43 44 55 64 75 77
Trace of Quicksort (cont.)

Left subarray has single value; it is sorted
| 12 | 23 | 33 | 43 | 44 | 55 | 64 | 75 | 77 |

Trace of Quicksort (cont.)
Algorithm for Quicksort

(We describe how to do the partitioning later)

- The iterators first and last are the end points of the array being sorted
- pivot is a pointer to the pivot value after partitioning

Algorithm for Quicksort

1. if the sequence has more than one element
2. Partition the elements in the sequence in the iterator range first through last so that the pivot value is in its correct place and is pointed to by pivot
3. Recursively apply quicksort to the sequence in the iterator range first through pivot
4. Recursively apply quicksort to the sequence in the iterator range pivot + 1 through last
Analysis of Quicksort

- If the pivot value is a random value selected from the current sequence,
  - then statistically, half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be log \( n \) levels of recursion
- At each recursion level, the partitioning process involves moving an element to its correct position—\( n \) moves
- Quicksort is \( O(n \log n) \), just like merge sort
The array split may not be the best case, i.e. 50-50

An exact analysis is difficult (and beyond the scope of this course), but, the running time will be bounded by a constant $\times n \log n$
A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty.

In that case, the sort will be $O(n^2)$.

Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts.

We’ll discuss a solution later.
Algorithm for Partitioning

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first.
Trace of Partitioning (cont.)

| 44 | 75 | 23 | 43 | 55 | 12 | 64 | 77 | 33 |

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript \textit{first}.
Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple.
For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple.
Search for the first value at the left end of the array that is greater than the pivot value.
Search for the first value at the left end of the array that is greater than the pivot value.
Then search for the first value at the right end of the array that is less than or equal to the pivot value.
Then search for the first value at the right end of the array that is less than or equal to the pivot value.
Trace of Partitioning (cont.)

Exchange these values

44 75 23 43 55 12 64 77 33
Trace of Partitioning (cont.)

Exchange these values
Trace of Partitioning (cont.)

Repeat
Trace of Partitioning (cont.)

Find first value at left end greater than pivot
Trace of Partitioning (cont.)

Find first value at right end less than or equal to pivot
Trace of Partitioning (cont.)

Exchange

44 33 23 43 12 55 64 77 75
Trace of Partitioning (cont.)

Repeat

44 33 23 43 12 55 64 77 75
Trace of Partitioning (cont.)

Find first element at left end greater than pivot
Trace of Partitioning (cont.)

Find first element at right end less than or equal to pivot
Since down has "passed" up, do not exchange
Trace of Partitioning (cont.)

Exchange the pivot value with the value at down
Trace of Partitioning (cont.)

Exchange the pivot value with the value at \textit{down}
The pivot value is in the correct position; return the value of `down` and assign it to the iterator `pivot`
Algorithm for partition Function

1. Define the pivot value as the contents of table[first].
2. Initialize up to first + 1 and down to last - 1.
3. do
4. Increment up until up selects the first element greater than the pivot value or up has reached last - 1.
5. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first.
6. if up < down then
7. Exchange table[up] and table[down].
8. while up is to the left of down
9. Exchange table[first] and table[down].
10. Return the value of down to pivot.
Algorithm for \textit{partition} Function (cont.)

\begin{itemize}
\item \texttt{pivot\_value} = 30
\item \texttt{pivot\_value} = 85
\end{itemize}

\textit{After swap}

\begin{itemize}
\item \texttt{pivot\_value} = 85
\end{itemize}
A Revised Partition Algorithm

- Quicksort is $O(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted.
- The worst possible performance occurs for a sorted array, which is not very desirable.
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split:
  - Use three references: first, middle, last.
  - Select the median of these items as the pivot.
A Revised Partition Algorithm (cont.)

Algorithm for Revised partition Function

1. Sort table[first], table[middle], and table[last – 1].
2. Move the median value to the first position (the pivot value) by exchanging table[first] and table[middle].
3. Initialize up to first + 1 and down to last – 1.
4. do
5. Increment up until up selects the first element greater than the pivot value or up has reached last – 1.
6. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first.
7. if up < down then
8. Exchange table[up] and table[down]
9. while up is to the left of down.
10. Exchange table[first] and table[down].
11. Return the value of down to pivot.
Trace of Revised Partitioning
Trace of Revised Partitioning (cont.)

```
+---+---+---+---+---+---+---+
| 44 | 75 | 23 | 43 | 55 | 12 | 64 |
+---+---+---+---+---+---+---+
```

- **first**: 44
- **middle**: 55
- **last**: 33
Trace of Revised Partitioning (cont.)

Sort these values
Trace of Revised Partitioning (cont.)

Sort these values:

- First: 33, 75, 23, 43
- Middle: 44, 12, 64
- Last: 77, 55
Trace of Revised Partitioning (cont.)

- Exchange middle with first

```
33 75 23 43
44 12 64 77 55
```
Trace of Revised Partitioning (cont.)

Exchange middle with first
Run the partition algorithm using the first element as the pivot
Algorithm for Revised partition Function

1. Sort table[first], table[middle], and table[last – 1]
2. Move the median value to the first position (the pivot value) by exchanging table[first] and table[middle]
3. Initialize up to first + 1 and down to last – 1
4. do
5. Increment up until up selects the first element greater than the pivot value or up has reached last – 1
6. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first
7. if up < down then
8. Exchange table[up] and table[down]
9. while up is to the left of down
10. Exchange table[first] and table[down]
11. Return the value of down to pivot
Comparison of Sort Algorithms

Summary
### Sort Review

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n^{7/6})$</td>
<td>$O(n^{5/4})$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

[http://bigocheatsheet.com/](http://bigocheatsheet.com/)