Sorting Midterm Exams
Sorting

- Input: 34 8 64 51 32 21

- Array containing unordered Comparables (duplicates allowed).

- Output: 8 21 32 34 51 64

- A sorted array containing the same items.

- Only comparisons between pairs of items allowed (comparison based sorting).
Sorting Applications
Sorting Applications

• Sorting email by date / subject …, Sorting files by name.
Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).
Sorting Applications

• Sorting email by date / subject …, Sorting files by name.

• Selection problem (find the k-th largest, find the median).

• Efficient search (binary search on sorted data).
Sorting Applications

• Sorting email by date / subject …, Sorting files by name.

• Selection problem (find the k-th largest, find the median).

• Efficient search (binary search on sorted data).

• Finding duplicates.
Sorting Applications

- Sorting email by date / subject …, Sorting files by name.
- Selection problem (find the k-th largest, find the median).
- Efficient search (binary search on sorted data).
- Finding duplicates.
- Greedy algorithms (explore k highest scoring paths first).
Sorting Applications

• Sorting email by date / subject …, Sorting files by name.

• Selection problem (find the k-th largest, find the median).

• Efficient search (binary search on sorted data).

• Finding duplicates.

• Greedy algorithms (explore k highest scoring paths first).

• …
Sorting Overview

• We will discuss different sorting algorithms and compare their running time, required space, and stability.

• Insertion sort

• Shell sort

• Heap sort

• Merge sort

• Quick sort

• Radix Sort
Insertion Sort

- Perform $p=1\ldots N-1$ passes through the array.
- Assume array[0..p-1] is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.
Insertion Sort

- Perform \( p=1 \ldots N-1 \) passes through the array.
- Assume array\([0..p-1]\) is already sorted.
- Take the element \( x \) at position \( p \).
- Repeatedly swap \( x \) its left neighbor until it is in the correct position.
**Insertion Sort**

- Perform $p=1\ldots N-1$ passes through the array.
- Assume array[0..p-1] is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.

$p=2$
Insertion Sort

- Perform $p=1\ldots N-1$ passes through the array.
- Assume array[0..p-1] is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.

$p=3$
Insertion Sort

- Perform \( p=1 \ldots N-1 \) passes through the array.
- Assume array[0..p-1] is already sorted.
- Take the element \( x \) at position \( p \).
- Repeatedly swap \( x \) its left neighbor until it is in the correct position.

\[
\begin{array}{cccccc}
8 & 34 & 51 & 64 & 32 & 21 \\
\end{array}
\]

\( p=3 \)
Insertion Sort

- Perform $p=1 \ldots N-1$ passes through the array.
- Assume $\text{array}[0..p-1]$ is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.
Insertion Sort

- Perform \( p = 1 \ldots N-1 \) passes through the array.
- Assume array\([0..p-1]\) is already sorted.
- Take the element \( x \) at position \( p \).
- Repeatedly swap \( x \) its left neighbor until it is in the correct position.
Insertion Sort

- Perform $p=1...N-1$ passes through the array.
- Assume array[0..$p-1$] is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ and its left neighbor until it is in the correct position.

$p=4$
Insertion Sort

• Perform \( p = 1 \ldots N-1 \) passes through the array.
• Assume array[0..p-1] is already sorted.
• Take the element \( x \) at position \( p \).
• Repeatedly swap \( x \) its left neighbor until it is in the correct position.
Insertion Sort

- Perform \( p = 1 \ldots N-1 \) passes through the array.
- Assume array\([0..p-1]\) is already sorted.
- Take the element \( x \) at position \( p \).
- Repeatedly swap \( x \) its left neighbor until it is in the correct position.
Insertion Sort

- Perform \( p = 1 \ldots N-1 \) passes through the array.
- Assume array[0..p-1] is already sorted.
- Take the element \( x \) at position \( p \).
- Repeatedly swap \( x \) its left neighbor until it is in the correct position.
Insertion Sort

- Perform $p=1\ldots N-1$ passes through the array.
- Assume array$[0..p-1]$ is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.
Insertion Sort

- Perform $p=1\ldots N-1$ passes through the array.
- Assume array[0..p-1] is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.
**Insertion Sort**

- Perform $p=1\ldots N-1$ passes through the array.
- Assume array$[0..p-1]$ is already sorted.
- Take the element $x$ at position $p$.
- Repeatedly swap $x$ its left neighbor until it is in the correct position.

```
8 21 32 34 51 64
```

$p=5$
Insertion Sort

```c
void insertionSort( Integer [ ] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```
Insertion Sort

```java
void insertionSort( Integer [ ] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[j - 1]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```

Total: $O(N^2)$
Insertion Sort

```java
void insertionSort( Integer [] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```

Total: $O(N^2)$

Best case input (sorted): $O(N)$
Insertion Sort

```java
void insertionSort( Integer [] a ) {
    int j;

    for( int p = 1; p < a.length; p++ ) {
        Integer x = a[ p ];
        for( j = p; j > 0 && x < a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = x;
    }
}
```

Total: $O(N^2)$

Best case input (sorted): $O(N)$

Worst case input (sorted in reverse order):

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

$$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[ h_3 = 5 \quad h_2 = 3 \quad h_1 = 1 \]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase k makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccc}
21 & 8 & 64 & 51 & 32 & 34 & 7 & 30 & 1 & 2 & 5 \\
\end{array}
\]

$h_3 = 5$   $h_2 = 3$   $h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccc}
21 & 7 & 64 & 51 & 32 & 34 & 8 & 30 & 1 & 2 & 5
\end{array}
\]

$h_3 = 5$  \hspace{0.5cm} h_2 = 3 \hspace{0.5cm} h_1 = 1
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

| 21 |  7 |  64 |  51 |  32 |  34 |  8 |  30 |  1 |  2 |  5 |

$h_3 = 5$  \hspace{1cm} h_2 = 3  \hspace{1cm} h_1 = 1
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

| 21 | 7 | 30 | 51 | 32 | 34 | 8 | 64 | 1 | 2 | 5 |

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase \( k \) makes sure that all items space \( h_k \) apart are sorted.

• “increment sequence” of steps \( h_1, h_2, \ldots, h_t \)

\[
\begin{array}{cccccccc}
21 & 7 & 30 & 51 & 32 & 34 & 8 & 64 & 1 & 2 & 5 \\
\end{array}
\]

\( h_3 = 5 \quad h_2 = 3 \quad h_1 = 1 \)
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

| 21 | 7 | 30 | 1 | 32 | 34 | 8 | 64 | 51 | 2 | 5 |

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccccc}
21 & 7 & 30 & 1 & 32 & 34 & 8 & 64 & 51 & 2 & 5 \\
\end{array}
\]

$h_3 = 5$  \hspace{1em} h_2 = 3  \hspace{1em} h_1 = 1
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{align*}
21 & \quad 7 & \quad 30 & \quad 1 & \quad 2 & \quad 34 & \quad 8 & \quad 64 & \quad 51 & \quad 32 & \quad 5 \\
\end{align*}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “Increment sequence” of steps $h_1$, $h_2$, …, $h_t$

\[
\begin{align*}
21 & \quad 7 & \quad 30 & \quad 1 & \quad 2 & \quad 34 & \quad 8 & \quad 64 & \quad 51 & \quad 32 & \quad 5 \\
\end{align*}
\]

\[
\begin{align*}
h_3 &= 5 & h_2 &= 3 & h_1 &= 1 \\
\end{align*}
\]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1$, $h_2$, … , $h_t$

\[
\begin{array}{cccccccccc}
21 & 7 & 30 & 1 & 2 & 5 & 8 & 64 & 51 & 32 & 34 \\
\end{array}
\]

$h_3 = 5$  \hspace{1em} h_2 = 3  \hspace{1em} h_1 = 1
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1$, $h_2$, $h_3$, ..., $h_t$

| 5 | 7 | 30 | 1 | 2 | 21 | 8 | 64 | 51 | 32 | 34 |

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

$$
egin{array}{cccccccccc}
1 & 7 & 30 & 5 & 2 & 21 & 8 & 64 & 51 & 32 & 34
\end{array}
$$

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- "increment sequence" of steps $h_1, h_2, \ldots , h_t$

1 | 7 | 30 | 5 | 2 | 21 | 8 | 64 | 51 | 32 | 34

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

| 1 | 2 | 30 | 5 | 7 | 21 | 8 | 64 | 51 | 32 | 34 |

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccc}
1 & 2 & 30 & 5 & 7 & 21 & 8 & 64 & 51 & 32 & 34 \\
\end{array}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase \( k \) makes sure that all items space \( h_k \) apart are sorted.

• “increment sequence” of steps \( h_1, h_2, \ldots, h_t \)

\[
\begin{array}{cccccccccccc}
1 & 2 & 21 & 5 & 7 & 30 & 8 & 64 & 51 & 32 & 34 \\
\end{array}
\]

\( h_3 = 5 \) \quad \textbf{h}_2 = 3 \quad h_1 = 1 \]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{align*}
1 & \quad 2 & \quad 21 & \quad 5 & \quad 7 & \quad 30 & \quad 8 & \quad 64 & \quad 51 & \quad 32 & \quad 34 \\
\end{align*}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1$, $h_2$, …, $h_t$

| 1 | 2 | 21 | 5 | 7 | 30 | 8 | 64 | 51 | 32 | 34 |

$h_3 = 5$  $h_2 = 3$  $h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>21</td>
<td>5</td>
<td>7</td>
<td>30</td>
<td>8</td>
<td>64</td>
<td>51</td>
<td>32</td>
</tr>
</tbody>
</table>

$h_3 = 5$  \hspace{1cm} h_2 = 3  \hspace{1cm} h_1 = 1
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>21</th>
<th>5</th>
<th>7</th>
<th>30</th>
<th>8</th>
<th>64</th>
<th>51</th>
<th>32</th>
<th>34</th>
</tr>
</thead>
</table>

$h_3 = 5$  \quad h_2 = 3  \quad h_1 = 1
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{align*}
1 & \quad 2 \quad 21 \quad 5 \quad 7 \quad 30 \quad 8 \quad 64 \quad 51 \quad 32 \quad 34 \\
\end{align*}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase k makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1$, $h_2$, … , $h_t$

\[
\begin{array}{cccccccccc}
1 & 2 & 21 & 5 & 7 & 30 & 8 & 34 & 51 & 32 & 64 \\
\end{array}
\]

$h_3 = 5$  $h_2 = 3$  $h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.
- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.
- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccccc}
1 & 2 & 21 & 5 & 7 & 30 & 8 & 34 & 51 & 32 & 64 \\
\end{array}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>21</th>
<th>5</th>
<th>7</th>
<th>30</th>
<th>8</th>
<th>34</th>
<th>51</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
</table>

$h_3 = 5$  $h_2 = 3$  $h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{align*}
1 & \quad 2 & \quad 5 & \quad 21 & \quad 7 & \quad 30 & \quad 8 & \quad 34 & \quad 51 & \quad 32 & \quad 64 \\
h_3 &= 5 \quad h_2 &= 3 \quad h_1 &= 1
\end{align*}
\]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase \( k \) makes sure that all items space \( h_k \) apart are sorted.

• “increment sequence” of steps \( h_1, h_2, \ldots, h_t \)

\[
\begin{array}{ccccccccccc}
1 & 2 & 5 & 21 & 7 & 30 & 8 & 34 & 51 & 32 & 64 \\
\end{array}
\]

\( h_3 = 5 \quad h_2 = 3 \quad h_1 = 1 \)
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “Increment sequence” of steps $h_1, h_2, \ldots, h_t$

- $h_3 = 5, \quad h_2 = 3, \quad h_1 = 1$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{align*}
1 & \quad 2 & \quad 5 & \quad 7 & \quad 21 & \quad 30 & \quad 8 & \quad 34 & \quad 51 & \quad 32 & \quad 64 \\
h_3 & = 5 & \quad h_2 & = 3 & \quad h_1 & = 1
\end{align*}
\]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

$$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$$
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1$, $h_2$, … , $h_t$

\[ h_3 = 5 \quad h_2 = 3 \quad h_1 = 1 \]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{align*}
h_3 & = 5 \\
h_2 & = 3 \\
h_1 & = 1
\end{align*}
\]
Shell Sort

- Generalize insertion sort so that items that are further apart can be swapped.

- Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

- “increment sequence” of steps $h_1, h_2, \ldots, h_t$

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>21</td>
<td>30</td>
<td>34</td>
<td>51</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “Increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccccc}
1 & 2 & 5 & 7 & 8 & 21 & 30 & 34 & 51 & 32 & 64 \\
\end{array}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[ \begin{align*}
1 & \quad 2 & \quad 5 & \quad 7 & \quad 8 & \quad 21 & \quad 30 & \quad 34 & \quad 32 & \quad 51 & \quad 64 \\
\end{align*} \]

\[ h_3 = 5 \quad h_2 = 3 \quad h_1 = 1 \]
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccccc}
1 & 2 & 5 & 7 & 8 & 21 & 30 & 32 & 34 & 51 & 64 \\
\end{array}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

• Generalize insertion sort so that items that are further apart can be swapped.

• Break up sorting into phases. Each phase $k$ makes sure that all items space $h_k$ apart are sorted.

• “increment sequence” of steps $h_1, h_2, \ldots, h_t$

\[
\begin{array}{cccccccccccc}
1 & 2 & 5 & 7 & 8 & 21 & 30 & 32 & 34 & 51 & 64 \\
\end{array}
\]

$h_3 = 5 \quad h_2 = 3 \quad h_1 = 1$
Shell Sort

- The running time analysis for shell sort is complex and depends on the specific increment sequence.
- With Hibbard’s sequence \((1,3,7,15,\ldots,2^k-1)\) worst case running time is \(\Theta(N^{3/2})\).
Sorting Stability

• Assume we put key/value pairs sorted by keys into the array.

• Shell Sort is *unstable*: keys will be sorted, but values for the same key may be in different order than in the input.
Sorting Stability

- Assume we put key/value pairs sorted by keys into the array.

- Shell Sort is *unstable*: keys will be sorted, but values for the same key may be in different order than in the input.
Space Requirements

• Both Insertion Sort and Shell Sort operate in place.

• Only a small amount of memory required to store a temporary value for swaps.

• Space requirement: $O(1)$
Heap Sort

- First convert an unordered array into a heap in \(O(N)\) time.
- Then perform \(N\) \texttt{deleteMin}\ operations to retrieve the elements in sorted order.
  - each \texttt{deleteMin} is \(O(\log N)\)
Heap Sort

• First convert an unordered array into a heap in O(N) time.

• Then perform N deleteMin operations to retrieve the elements in sorted order.
  
  • each deleteMin is O(log N)

• Problem: This algorithm requires a second array to store the output: O(N) space!

• Idea: re-use the freed space after each deleteMin.
Heap Sort Example
Heap Sort Example
Heap Sort Example

Build heap in O(N) time
Heap Sort Example

Build heap in $O(N)$ time

```
1  2  3  7  4  8  6  10  9  5  11
```
Heap Sort Example

deleteMin, write min element into empty cell

```
| 1 | 2 | 3 | 7 | 4 | 8 | 6 | 10 | 9 | 5 | 11 |
```

```plaintext
1 1 3 8 6 10 9 5 2 7 4 11
```
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

Percolate down

2 4 3 7 5 8 6 10 9 11 1
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

Percolate down

3 4 6 7 5 8 11 10 9 2 1
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

deleteMin, write min element into empty cell

\[
\begin{array}{c}
9 \\
10 \\
11 \\
\end{array}
\]

\[
\begin{array}{c}
9 \\
10 \\
11 \\
\end{array}
\]

\[
\begin{array}{c c c c c c c c c c}
9 & 10 & 11 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
\end{array}
\]
Heap Sort Example

deleteMin, write min element into empty cell
Heap Sort Example

- Can use a max-heap if we want the output in increasing order.
Merge Sort

• A classic *divide-and-conquer* algorithm.

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

```
34  8  64  2  51  32  21  1
```
Merge Sort

• A classic *divide-and-conquer* algorithm.

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

<table>
<thead>
<tr>
<th>34</th>
<th>8</th>
<th>64</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>
Merge Sort

• A classic divide-and-conquer algorithm.

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

<table>
<thead>
<tr>
<th>2</th>
<th>8</th>
<th>34</th>
<th>64</th>
</tr>
</thead>
</table>

| 1 | 21 | 32 | 51 |
Merge Sort

• A classic *divide-and-conquer* algorithm.

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

\[
\begin{array}{cccc}
2 & 8 & 34 & 64 \\
\end{array}
\quad
\begin{array}{cccc}
1 & 21 & 32 & 51 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
1 & 2 & 8 & 21 & 32 & 34 & 51 & 64 \\
\end{array}
\]
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If \( a[\text{Actr}] < a[\text{Bctr}] \), copy \( a[\text{Actr}] \) to tmp and advance Actr.
  - Otherwise, copy \( a[\text{Bctr}] \) to the output and advance Bctr.
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $a[\text{Actr}] < a[\text{Bctr}]$, copy $a[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $a[\text{Bctr}]$ to the output and advance Bctr.

```
<table>
<thead>
<tr>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 8 34 64</td>
<td>1 21 32 51</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>tmp</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Actr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Bctr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Cctr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

52
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $A[\text{Actr}] < A[\text{Bctr}]$, copy $A[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $A[\text{Bctr}]$ to the output and advance Bctr.

```plaintext
a

2  8  34  64

Bctr

1  21  32  51

Actr

tmp

1  2

Cctr

52
```
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $a[\text{Actr}] < a[\text{Bctr}]$, copy $a[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $a[\text{Bctr}]$ to the output and advance Bctr.

```
<table>
<thead>
<tr>
<th>a</th>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 8 34 64</td>
<td>1 2</td>
</tr>
</tbody>
</table>
```

$\text{Actr}$ $\text{Bctr}$ $\text{Cctr}$
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $a[\text{Actr}] < a[\text{Bctr}]$, copy $a[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $a[\text{Bctr}]$ to the output and advance Bctr.

```
a
  2  8  34  64
  ↑   Actr

tmp
  1  2  8  21
  ↑   Cctr

  1  21  32  51
  ↑   Bctr
```
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $a[\text{Actr}] < a[\text{Bctr}]$, copy $a[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $a[\text{Bctr}]$ to the output and advance Bctr.

```
a
\begin{array}{cccc}
2 & 8 & 34 & 64 \\
\end{array}
```
```
b
\begin{array}{cccc}
1 & 21 & 32 & 51 \\
\end{array}
```
```
tmp
\begin{array}{cccc}
1 & 2 & 8 & 21 & 32 \\
\end{array}
```
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $a[\text{Actr}] < a[\text{Bctr}]$, copy $a[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $a[\text{Bctr}]$ to the output and advance Bctr.
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If $a[\text{Actr}] < a[\text{Bctr}]$, copy $a[\text{Actr}]$ to tmp and advance Actr.
  - Otherwise, copy $a[\text{Bctr}]$ to the output and advance Bctr.

\[
\begin{array}{cccc}
2 & 8 & 34 & 64 \\
\end{array}
\qquad
\begin{array}{cccc}
1 & 21 & 32 & 51 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 2 & 8 & 21 & 32 & 34 & 51 & \text{tmp} \\
\end{array}
\]

Actr

Bctr

Cctr
Merging Sorted Sublists

- Keep a pointers for each sub-list in the array.
- In each step, compare the elements they point two.
  - If \text{a}[\text{Actr}] < \text{a}[\text{Bctr}]\), copy \text{a}[\text{Actr}] to tmp and advance Actr.
  - Otherwise, copy \text{a}[\text{Bctr}] to the output and advance Bctr.

\[
\begin{array}{c|c|c|c|c}
\text{a} & 2 & 8 & 34 & 64 \\
\hline
\end{array}
\begin{array}{c|c|c|c|c}
\text{tmp} & 1 & 2 & 8 & 21 & 32 & 34 & 51 & 64 \\
\hline
\end{array}
\begin{array}{c|c|c|c|c}
\text{1} & 21 & 32 & 51 & 64 \\
\hline
\end{array}
\]
private static <T extends Comparable<T>>
void merge( T[] a, T[] tmpArray, int aCtr, int bCtr, int rightEnd ) {
    int leftEnd = bCtr - 1;
    int tmpPos = aCtr;
    int numElements = rightEnd - aCtr + 1;

    // Main loop
    while( aCtr <= leftEnd && bCtr <= rightEnd )
    {
        if( a[ aCtr ].compareTo( a[ bCtr ] ) <= 0 )
            tmpArray[ tmpPos++ ] = a[ aCtr++ ];
        else
            tmpArray[ tmpPos++ ] = a[ bCtr++ ];

        while( aCtr <= leftEnd )  // Copy rest of first half
            tmpArray[ tmpPos++ ] = a[ aCtr++ ];

        while( bCtr <= rightEnd )  // Copy rest of right half
            tmpArray[ tmpPos++ ] = a[ bCtr++ ];

    }

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}
Merge Sort

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

| 34 | 8  | 64 | 2  | 51 | 32 | 21 | 1  |
Merge Sort

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

\[
\begin{array}{cccc}
34 & 8 & 64 & 2 \\
\end{array}
\quad
\begin{array}{cccc}
51 & 32 & 21 & 1 \\
\end{array}
\]
Merge Sort

- Split the array in half, recursively sort each half.
- Merge the two sorted lists.
Merge Sort

• Split the array in half, recursively sort each half.

• Merge the two sorted lists.

```
34 8  64 2  51 32  21 1
8 34  2 64  32 51  1 21
```
Merge Sort

- Split the array in half, recursively sort each half.

- Merge the two sorted lists.

```
<table>
<thead>
<tr>
<th>34</th>
<th>8</th>
<th>64</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>34</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>34</td>
<td>64</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>51</th>
<th>32</th>
<th>21</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>51</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>32</td>
<td>51</td>
</tr>
</tbody>
</table>
```
Merge Sort

- Split the array in half, recursively sort each half.

- Merge the two sorted lists.
Merge Sort - Implementation

```java
private static <T extends Comparable<T>>
void mergeSort( T[] a, T[] tmpArray, int left, int right )
{
    if( left < right ) {
        int center = (left + right) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
```
Merge Sort Running Time

• This running time analysis is typical for divide and conquer algorithms.

• Merge sort is a recursive algorithm. The running time analysis should be similar to what we have seen for other algorithms of this type (e.g. binary search)

• Base case: N=1 (sort a 1-element list). T(1) = 1

• Recurrence: T(N) = 2 T(N/2) + N
Merge Sort Running Time

\[
T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N
\]

\[
= 2 \cdot (2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2}) + N = 4 \cdot T\left(\frac{N}{4}\right) + N + N
\]

\[
= 2^k \cdot T\left(\frac{N}{2^k}\right) + k \cdot N
\]

assumed \( k = \log N \)

\[
= N \cdot T(1) + \log N \cdot N
\]

\[
= N + N \cdot \log N = \Theta(N \log N)
\]
Merge Sort Properties

• Worst case running time: $\Theta(N \log N)$

• Is MergeSort stable?

• Space requirement?
Merge Sort Properties

- Worst case running time: $\Theta(N \log N)$
- Is MergeSort stable?
  Yes. Merging preserves order of elements.
- Space requirement?
Merge Sort Properties

• Worst case running time: $\Theta(N \log N)$

• Is MergeSort stable?
  Yes. Merging preserves order of elements.

• Space requirement?
  Need a temporary array. $O(N)$