COMP2521 23T3
Sorting Algorithms (I)

Kevin Luxa
cs2521@cse.unsw.edu.au

properties of sorting algorithms
elementary sorting algorithms
• Sorting enables faster searching
  • Binary search

• Sorting arranges data in useful ways (for humans and computers)
  • For example, a list of students in a tutorial

• Sorting provides a useful intermediate for other algorithms
  • For example, duplicate detection/removal, merging two collections
• Sorting involves arranging a collection of items in order
  • Arrays, linked lists, files
• Items are sorted based on some property (called the key), using an ordering relation on that property
  • Numbers are sorted numerically
  • Strings are sorted alphabetically
We sort arrays of Items, which could be:

- Simple values: int, char, double
- Complex values: strings
- Structured values: struct

The items are sorted based on a key, which could be:

- The entire item, if the item is a single value
- One or more fields, if the item is a struct
Example: Each student has an ID and a name

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5151515</td>
<td>John</td>
</tr>
<tr>
<td>5012345</td>
<td>Jane</td>
</tr>
<tr>
<td>3456789</td>
<td>Bob</td>
</tr>
<tr>
<td>5050505</td>
<td>Alice</td>
</tr>
<tr>
<td>5555555</td>
<td>John</td>
</tr>
<tr>
<td>5432109</td>
<td>Andrew</td>
</tr>
</tbody>
</table>

Sorting by ID (i.e., key is ID):

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3456789</td>
<td>Bob</td>
</tr>
<tr>
<td>5012345</td>
<td>Jane</td>
</tr>
<tr>
<td>5050505</td>
<td>Alice</td>
</tr>
<tr>
<td>5151515</td>
<td>John</td>
</tr>
<tr>
<td>5432109</td>
<td>Andrew</td>
</tr>
<tr>
<td>5555555</td>
<td>John</td>
</tr>
</tbody>
</table>

Sorting by name (i.e., key is name):

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5050505</td>
<td>Alice</td>
</tr>
<tr>
<td>5432109</td>
<td>Andrew</td>
</tr>
<tr>
<td>3456789</td>
<td>Bob</td>
</tr>
<tr>
<td>5012345</td>
<td>Jane</td>
</tr>
<tr>
<td>5151515</td>
<td>John</td>
</tr>
<tr>
<td>5555555</td>
<td>John</td>
</tr>
</tbody>
</table>
Arrange items in array slice $a[lo..hi]$ into sorted order:

```
| | | | | ... | | | | |
```

To sort an entire array of size $N$, $lo == 0$ and $hi == N - 1$. 

```
| | | | | ... | | | | |
```

$unordered$

$sort(a, lo, hi)$

```
| | | | | ... | | | | |
```

$ordered$
Pre-conditions:
array $a[N]$ of Items
lo, hi are valid indices on $a$
(roughly, $0 \leq lo < hi \leq N - 1$)

Post-conditions:
array $a[lo..hi]$ contains the same values as before the sort
$a[lo] \leq a[lo + 1] \leq a[lo + 2] \leq \ldots \leq a[hi]$
Properties:

- Stability
- Adaptability
- In-place
A stable sort preserves the relative order of items with equal keys.

Formally: For all pairs of items $x$ and $y$ where $\text{KEY}(x) \equiv \text{KEY}(y)$, if $x$ precedes $y$ in the original array, then $x$ precedes $y$ in the sorted array.
Example: Each card has a value and a suit

A stable sort on value:

![Image of playing cards demonstrating stability in sorting algorithms]
Example: Each card has a value and a suit

Example of an unstable sort on value:
When is stability important?

- When sorting the same array multiple times on different keys
- Some sorting algorithms rely on this, for example, radix sort
Example: Array of first names and last names

Sort by last name:

Then sort by first name (using stable sort):
Stability doesn’t matter if...

- All items have unique keys
  - Example: Sorting students by ID
- The key is the entire item
  - Example: Sorting an array of integer values
Properties of Sorting Algorithms

Adaptability

- An **adaptive** sorting algorithm takes advantage of existing order in its input
  - Time complexity of an adaptive sorting algorithm will be better for sorted or nearly-sorted inputs
- Can be a useful property, depending on whether nearly sorted inputs are common
• An in-place sorting algorithm sorts the data within the original structure, without using temporary arrays
// we deal with generic `Item's

typedef int Item;

// abstractions to hide details of items
#define key(A) (A)
#define lt(A, B) (key(A) < key(B))
#define le(A, B) (key(A) <= key(B))
#define ge(A, B) (key(A) >= key(B))
#define gt(A, B) (key(A) > key(B))

// Sort a slice of an array of Items
void sort(Item a[], int lo, int hi);
This framework can be adapted by...
defining a different data structure for Item;
defining a method for extracting sort keys;
defining a different ordering (less);
defining a different swap method for different Item

typedef struct {
    char *name;
    char *course;
} Item;

#define key(A) (A.name)
#define lt(A, B) (strcmp(key(A), key(B)) < 0)
#define le(A, B) (strcmp(key(A), key(B)) <= 0)
#define ge(A, B) (strcmp(key(A), key(B)) >= 0)
#define gt(A, B) (strcmp(key(A), key(B)) > 0)
Analysis of Sorting Algorithms

In analysing sorting algorithms:

- \( n \): the number of items \((hi - lo + 1)\)
- \( C \): the number of comparisons between items
- \( S \): the number of times items are swapped

(We usually aim to minimise \( C \) and \( S \).)

Cases to consider for input order:

- random order: Items in \([lo..hi]\) have no ordering
- sorted order: \(a[lo] \leq a[lo + 1] \leq \cdots \leq a[hi]\)
- reverse-sorted order: \(a[lo] \geq a[lo + 1] \geq \cdots \geq a[hi]\)
Examples of Sorting Algorithms

Elementary sorting algorithms:
- Selection sort
- Bubble sort
- Insertion sort
- Shell sort

More efficient sorting algorithms:
- Merge sort
- Quick sort

Non-comparison-based sorting algorithms:
- Radix sort
- Key-indexed counting sort
Method:
- Find the smallest element, swap it with the first element
- Find the second-smallest element, swap it with the second element
- ...
- Find the second-largest element, swap it with the second-last element

Each iteration improves the “sortedness” of the array by one element
Selection Sort Example

\[\begin{array}{cccccccc}
4 & 1 & 7 & 3 & 8 & 6 & 5 & 2 \\
\end{array}\]
Selection Sort Example
Selection Sort

Example

1 4 7 3 8 6 5 2

1 4 7 3 8 6 5 2

1 2 7 3 8 6 5 4
Selection Sort

Example
Selection Sort

Example

The diagram illustrates the Selection Sort algorithm on a list of numbers. The algorithm works by repeatedly finding the minimum element from the unsorted part and putting it at the beginning. The process is repeated for the remaining unsorted part until the whole list is sorted.

Here is the step-by-step process:

Step 1:

4 1 7 3 8 6 5 2

Step 2:

1 4 7 3 8 6 5 2

Step 3:

1 2 7 3 8 6 5 4

Step 4:

1 2 3 7 8 6 5 4

Step 5:

1 2 3 4 8 6 5 7
Selection Sort

Example

1 4 7 3 8 6 5 2
1 4 7 3 8 6 5 2
1 2 7 3 8 6 5 4
1 2 3 7 8 6 5 4
1 2 3 4 8 6 5 7
1 2 3 4 5 6 8 7
Selection Sort

Example

4 1 7 3 8 6 5 2
1 4 7 3 8 6 5 2
1 2 7 3 8 6 5 4
1 2 3 7 8 6 5 4
1 2 3 4 8 6 5 7
1 2 3 4 5 6 8 7
1 2 3 4 5 6 8 7
Selection Sort

Example

4  1  7  3  8  6  5  2
1  4  7  3  8  6  5  2
1  2  7  3  8  6  5  4
1  2  3  7  8  6  5  4
1  2  3  4  8  6  5  7
1  2  3  4  8  6  5  7
1  2  3  4  5  6  8  7
1  2  3  4  5  6  8  7
1  2  3  4  5  6  7  8
void selectionSort(Item items[], int lo, int hi) {
    for (int i = lo; i < hi; i++) {
        int min = i;
        for (int j = i + 1; j <= hi; j++) {
            if (lt(items[j], items[min])) {
                min = j;
            }
        }
        swap(items, i, min);
    }
}
Cost analysis:

- In the first iteration, \( n - 1 \) comparisons, 1 swap
- In the second iteration, \( n - 2 \) comparisons, 1 swap
- ...
- In the final iteration, 1 comparison, 1 swap

\[
C = (n - 1) + (n - 2) + \ldots + 1 = \frac{1}{2} n(n - 1) \Rightarrow O(n^2)
\]

\[
S = n - 1
\]

Cost is the same, regardless of the sortedness of the original array.
Selection sort is unstable

- Due to long-range swaps
- Example:
Selection Sort
Properties

Unstable
Due to long-range swaps

Non-adaptive
Performs same steps, regardless of sortedness of original array

In-place
Sorting is done within original array; does not use temporary arrays
Bubble Sort

Method:

- Make multiple passes from left (lo) to right
- On each pass, swap any out-of-order adjacent pairs
- Elements “bubble up” until they meet a larger element
- Stop if there are no swaps during a pass
  - This means the array is sorted
Bubble Sort

Example

4 3 6 1 2 5
Motivation
Overview
Selection Sort
Bubble Sort
Example
Implementation
Analysis
Properties
Insertion Sort
Shell Sort
Summary
Sorting Lists
Bubble Sort

Example

Second pass

3 4 1 2 5 6

3 4 1 2 5 6

3 1 4 2 5 6

3 1 2 4 5 6

3 1 2 4 5 6
Bubble Sort

Third pass

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Bubble Sort

Example

Fourth pass

No swaps made; stop
void bubbleSort(Item items[], int lo, int hi) {
    for (int i = hi; i > lo; i--) {
        bool swapped = false;
        for (int j = lo; j < i; j++) {
            if (gt(items[j], items[j + 1])) {
                swap(items, j, j + 1);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
Best case: Array is sorted

- Only a single pass required
- \( n - 1 \) comparisons, no swaps
- Best-case time complexity: \( O(n) \)
Worst case: Array is reverse-sorted

- $n - 1$ passes required
  - First pass: $n - 1$ comparisons
  - Second pass: $n - 2$ comparisons
  - ...
  - Final pass: 1 comparison

- Total comparisons: $(n - 1) + (n - 2) + \ldots + 1 = \frac{1}{2}n(n - 1)$
- Every comparison leads to a swap $\Rightarrow \frac{1}{2}n(n - 1)$ swaps
- Worst-case time complexity: $O(n^2)$
**Average-case time complexity:** $O(n^2)$

- Can show empirically by generating random sequences and sorting them
**Stable**
Comparisons are between adjacent elements only.
Elements are only swapped if out of order.

**Adaptive**
Bubble sort is $O(n^2)$ on average, $O(n)$ if input array is sorted.

**In-place**
Sorting is done within original array; does not use temporary arrays.
Insertion Sort

Method:

- Take first element and treat as sorted array (of length 1)
- Take next element and insert into sorted part of array so that order is preserved
  - This increases the length of the sorted part by one
- Repeat for remaining elements
Insertion Sort
Example

4 1 7 3 8 6 5 2
**Insertion Sort**

Example

```
4 1 7 3 8 6 5 2
1 4 7 3 8 6 5 2
```

Motivation

Overview

Selection Sort

Bubble Sort

Insertion Sort

Example

Implementation

Analysis

Properties

Shell Sort

Summary

Sorting Lists
Insertion Sort

Example

1 4 7 3 8 6 5 2

1 4 7 3 8 6 5 2

1 4 7 3 8 6 5 2
Insertion Sort

Example
Insertion Sort

Example
Insertion Sort Example

Original List:

4 1 7 3 8 6 5 2

Sorted List:

1 3 4 6 7 8 5 2

Steps:

1. Insertion:
   - 4 is inserted between 1 and 7.

2. Insertion:
   - 1 is inserted between 4 and 7.

3. Insertion:
   - 7 is inserted between 3 and 8.

4. Insertion:
   - 3 is inserted between 4 and 7.

5. Insertion:
   - 8 is inserted between 6 and 5.

6. Insertion:
   - 6 is inserted between 5 and 2.

7. Insertion:
   - 5 is inserted between 6 and 2.

8. Insertion:
   - 2 is inserted between 5 and the beginning of the list.
Insertion Sort

Example

4 1 7 3 8 6 5 2
1 4 7 3 8 6 5 2
1 4 7 3 8 6 5 2
1 3 4 7 8 6 5 2
1 3 4 7 8 6 5 2
1 3 4 6 7 8 5 2
1 3 4 5 6 7 8 2
Insertion Sort

Example

1 2 3 4 5 6 7 8

4 1 7 3 8 6 5 2

1 4 7 3 8 6 5 2

1 4 7 3 8 6 5 2

1 3 4 7 8 6 5 2

1 3 4 7 8 6 5 2

1 3 4 6 7 8 5 2

1 3 4 5 6 7 8 2

1 2 3 4 5 6 7 8
```c
void insertionSort(Item items[], int lo, int hi) {
    for (int i = lo + 1; i <= hi; i++) {
        Item item = items[i];
        int j = i;
        for (; j > lo && lt(item, items[j - 1]); j--) {
            items[j] = items[j - 1];
        }
        items[j] = item;
    }
}
```
Best case: Array is sorted

- Inserting each element requires one comparison
- \( n - 1 \) comparisons
- Best-case time complexity: \( O(n) \)

1 2 3 4 5 6 7 8
Worst case: Array is reverse-sorted

- Inserting $i$-th element requires $i$ comparisons
  - Inserting index 1 element requires 1 comparison
  - Inserting index 2 element requires 2 comparisons
  - ...
- Total comparisons: $1 + 2 + \ldots + (n - 1) = \frac{1}{2}n(n - 1)$
- Worst-case time complexity: $O(n^2)$
Average-case time complexity: $O(n^2)$

- Can show empirically by generating random sequences and sorting them
Insertion Sort

Properties

**Stable**
Elements are always inserted to the right of any equal elements.

**Adaptive**
Insertion sort is $O(n^2)$ on average, $O(n)$ if input array is sorted.

**In-place**
Sorting is done within original array; does not use temporary arrays.
Bubble sort and insertion sort really only consider adjacent elements.

If we make longer-distance exchanges, can we be more efficient?

What if we consider elements that are some distance apart?

Shell sort, invented by Donald Shell
Idea:

- An array is $h$-sorted if taking every $h$-th element yields a sorted array
- An $h$-sorted array is made up of $\frac{n}{h}$ interleaved sorted arrays
- Shell sort: $h$-sort the array for progressively smaller $h$, ending with $h = 1$
Example of $h$-sorted arrays:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>3-sorted</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>2-sorted</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>1-sorted</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>unsorted</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Shell Sort Example
### Shell Sort

#### Example

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>( h = 3 ) passes</td>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-sorted</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
### Shell Sort

#### Example

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$h = 3$ passes</td>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-sorted</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>$h = 2$ passes</td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>4</td>
<td></td>
<td>7</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2-sorted</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
### Shell Sort Example

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsorted</strong></td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td><strong>h = 3 passes</strong></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3-sorted</strong></td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td><strong>h = 2 passes</strong></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>4</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2-sorted</strong></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td><strong>h = 1 pass</strong></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
void shellSort(Item items[], int lo, int hi)
{
    int size = hi - lo + 1;
    // find appropriate h-value to start with
    int h;
    for (h = 1; h <= (size - 1) / 9; h = (3 * h) + 1);

    for (; h > 0; h /= 3) {
        for (int i = lo + h; i <= hi; i++) {
            Item item = items[i];
            int j = i;
            for (; j >= lo + h && lt(item, items[j - h]); j -= h) {
                items[j] = items[j - h];
            }
            items[j] = item;
        }
    }
}
• Efficiency of shell sort depends on the \( h \)-sequence
• Effective \( h \)-sequences have been determined empirically
• Many \( h \)-sequences have been found to be \( O(n^{3/2}) \)
  • For example: 1, 4, 13, 40, 121, 364, 1093, ...
  • \( h_{i+1} = 3h_i + 1 \)
• Some \( h \)-sequences have been found to be \( O(n^{4/3}) \)
  • For example: 1, 8, 23, 77, 281, 1073, 4193, ...
Shell Sort

Properties

Unstable
Due to long-range swaps

Adaptive
Shell sort applies a generalisation of insertion sort
(which is adaptive)

In-place
Sorting is done within original array; does not use temporary arrays
<table>
<thead>
<tr>
<th>Sort Type</th>
<th>Time complexity</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</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>
<tr>
<td>Shell sort</td>
<td>depends</td>
<td>depends</td>
</tr>
</tbody>
</table>
 Aside: Sorting Linked Lists

Selection sort:
- Let \( L = \) original list, \( S = \) sorted list (initially empty)
- Repeat the following until \( L \) is empty:
  - Find the node \( V \) containing the largest value in \( L \), and unlink it
  - Insert \( V \) at the front of \( S \)

Bubble sort:
- Traverse the list, comparing adjacent values
  - If value in current node is greater than value in next node, swap values
- Repeat the above until no swaps required in one traversal

Insertion sort:
- Let \( L = \) original list, \( S = \) sorted list (initially empty)
- For each node in \( L \):
  - Insert the node into \( S \) in order
Shell sort:

- Difficult to implement efficiently
- Can’t access specific index in constant time
  - Have to traverse from the beginning
https://forms.office.com/r/aPF09YHZ3X