COMP2521 24T2
Sorting Algorithms (II)
Elementary Sorting Algorithms

Sim Mautner
cs2521@cse.unsw.edu.au

Slides adapted from those by Kevin Luxa 2521 24T1
Selection 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

Here is an example of a list to be sorted:

4 1 7 3 8 6 5 2

After applying Selection Sort, the list would be sorted in ascending order.
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 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
- For example, sort these cards by value:
**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
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
Example

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

Bubble Sort

Example

Implementation

Analysis

Properties

Insertion Sort

Shell Sort

Summary

Sorting Lists

Appendix
Selection Sort
Bubble Sort
Example
Implementation
Analysis
Properties
Insertion Sort
Shell Sort
Summary
Sorting Lists
Appendix

Bubble Sort Example

Third pass

1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5 6
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: $O(n^2)$

- It can be proven that for a randomly ordered array, bubble sort needs to perform $\frac{1}{4}n(n - 1)$ swaps on average $\Rightarrow O(n^2)$
  - See appendix for details
- 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
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
Example

4 1 7 3 8 6 5 2
Insertion Sort
Example

@arrangement
4 1 7 3 8 6 5 2
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
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: \( O(n) \)

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]
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: $O(n^2)$

- Same reason as for bubble sort
- Can show empirically by generating random sequences and sorting them
**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 move elements by shifting them up/down one space at a time.

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></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
<td>8</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>
<td>8</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>
<td>9</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>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></td>
<td>7</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>
<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>1</td>
<td></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>
<tr>
<td>$h = 1$ pass</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
## Summary of Elementary Sorts

<table>
<thead>
<tr>
<th></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
Aside: Sorting Linked Lists

Shell sort:

- Difficult to implement efficiently
- Can’t access specific index in constant time
  - Have to traverse from the beginning
Summary

Sorting Lists

Appendix

https://forms.office.com/r/riGKCze1cQ
Appendix
New concept: inversion

An inversion is a pair of elements from a sequence where the left element is greater than the right element.

For example, consider the following array:

4 2 1 5 3

The array contains 5 inversions:
(4, 2), (4, 1), (4, 3), (2, 1), (5, 3)
Observation:

- In bubble sort, every swap reduces the number of inversions by 1.

The goal of the proof: Show that the average number of inversions in a randomly sorted array is $O(n^2)$.

- This implies the number of swaps required by bubble sort is $O(n^2)$ ...
- Which implies that the average-case time complexity of bubble sort is $O(n^2)$ or slower
  - (but we know that it can’t be slower than $O(n^2)$ since the worst-case time complexity of bubble sort is $O(n^2)$)
In a randomly sorted array:

- The minimum possible number of inversions is 0 (sorted array)
- The maximum possible number of inversions is $\frac{1}{2}n(n - 1)$ (reverse-sorted array)
Let $k$ be the number of inversions in a random permutation. By reversing this permutation, one can obtain a permutation with $\frac{1}{2} n(n - 1) - k$ inversions.

For example, suppose $n = 5$:

<table>
<thead>
<tr>
<th>Original</th>
<th>Reverse</th>
<th>Inversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 2 4 1 5</td>
<td>5 1 4 2 3</td>
<td>6</td>
</tr>
<tr>
<td>1 3 4 5 2</td>
<td>2 5 4 3 1</td>
<td>7</td>
</tr>
<tr>
<td>1 2 3 4 5</td>
<td>5 4 3 2 1</td>
<td>10</td>
</tr>
</tbody>
</table>
Thus, if we take all the possible permutations of an array and pair each permutation with its reverse, the total number of inversions in each pair is \( \frac{1}{2} n(n - 1) \).

This implies that the average number of inversions across all permutations is \( \frac{1}{4} n(n - 1) \), which is \( O(n^2) \).