Sorting

CSC 212: Data Structures

King Saud University

Outline

- The sorting problem.
- Properties of sorting algorithms.
- General purpose sorting algorithms:
  - Quadratic sorting algorithms: selection sort and bubble sort.
  - Sub-quadratic sorting algorithms: merge sort.

The sorting problem

Given a list of n totally orderable items, rearrange the list in increasing (or decreasing) order.

- Totally orderable means that any two items can be compared: numbers, characters, strings etc.
- The items are usually called *keys*.

The sorting problem

Given a list of n totally orderable items, rearrange the list in increasing (or decreasing) order.

- Totally orderable means that any two items can be compared: numbers, characters, strings etc.
- The items are usually called *keys*.

Example

This is an instance of the sorting problem:

$$(12, 5, 8, 16, 9, 31) \rightarrow (5, 8, 9, 12, 16, 31)\,.$$

The sorting problem

Given a list of n totally orderable items, rearrange the list in increasing (or decreasing) order.

- Totally orderable means that any two items can be compared: numbers, characters, strings etc.
- The items are usually called *keys*.

Example

This is an instance of the sorting problem:

$$(12, 5, 8, 16, 9, 31) \to (5, 8, 9, 12, 16, 31) \,.$$

Often, the elements to be sorted contain keys and data:

$$((5, A), (2, C), (7, A), (2, B), (1, B)) \to ((1, B), (2, C), (2, B), (5, A), (7, A)) \,.$$

Properties of sorting algorithms

- Time complexity: worst case and average case.

Properties of sorting algorithms

- Time complexity: worst case and average case.
- Space complexity: the amount of extra space needed by the algorithm.

Properties of sorting algorithms

- Time complexity: worst case and average case.
- Space complexity: the amount of extra space needed by the algorithm.

Definition

An algorithm is said *in-place* if it does not require more than $O(\log n)$ space in addition to the input.

Properties of sorting algorithms

- Time complexity: worst case and average case.
- Space complexity: the amount of extra space needed by the algorithm.

Definition

An algorithm is said *in-place* if it does not require more than $O(\log n)$ space in addition to the input.

- Stability: A sorting algorithm is *stable* if it does not change the order of equal elements.

Properties of sorting algorithms

- Time complexity: worst case and average case.
- Space complexity: the amount of extra space needed by the algorithm.

Definition

An algorithm is said *in-place* if it does not require more than $O(\log n)$ space in addition to the input.

- Stability: A sorting algorithm is *stable* if it does not change the order of equal elements.

Example

Given the input array: $\{(5, A), (2, C), (7, A), (2, B), (1, B)\}$, where we want to sort according to the first element of the pairs (the integers), then:

- Time complexity: worst case and average case.
- Space complexity: the amount of extra space needed by the algorithm.

Definition

An algorithm is said *in-place* if it does not require more than $O(\log n)$ space in addition to the input.

- Stability: A sorting algorithm is *stable* if it does not change the order of equal elements.

Example

Given the input array: $\{(5, A), (2, C), (7, A), (2, B), (1, B)\}$, where we want to sort according to the first element of the pairs (the integers), then:

- $\{(1, B), (2, C), (2, B), (5, A), (7, A)\}$ is a stable sorting.

Properties of sorting algorithms

- Time complexity: worst case and average case.
- Space complexity: the amount of extra space needed by the algorithm.

Definition

An algorithm is said *in-place* if it does not require more than $O(\log n)$ space in addition to the input.

- Stability: A sorting algorithm is *stable* if it does not change the order of equal elements.

Example

Given the input array: $\{(5, A), (2, C), (7, A), (2, B), (1, B)\}$, where we want to sort according to the first element of the pairs (the integers), then:

- $\{(1, B), (2, C), (2, B), (5, A), (7, A)\}$ is a stable sorting.
- $\{(1, B), (2, B), (2, C), (5, A), (7, A)\}$ is not a stable sorting, since the order of $(2, B)$ and $(2, C)$ is reversed.

General purpose sorting algorithms

- Algorithms which can be used to sort any type of keys.
- They are based on comparison only and do not assume any other property in the keys (for example, they do not require the keys to be integers or strings).

Selection sort

Selection sort gradually builds the sorted array by finding the correct key for each new position.

```java
public static void selectionSort(int[] A, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) { // Search for the minimum
            if (A[j] < A[min])
                min = j;
        }
        // Swap A[i] with A [min]
        int tmp = A[i];
        A[i] = A[min];
        A[min] = tmp;
    }
}
```

Example

$\Uparrow$ indicates i, $\uparrow$ indicates min.

$$\left(\underset{\Uparrow}{12}, \underset{\uparrow}{5}, 8, 16, 9, 31\right)$$

Selection sort

Example

$\Uparrow$ indicates i, $\uparrow$ indicates min.

$$\left(\underset{\Uparrow}{12}, \underset{\uparrow}{5}, 8, 16, 9, 31\right)$$

$$\left(5, \underset{\Uparrow}{12}, \underset{\uparrow}{8}, 16, 9, 31\right)$$

Example

$\Uparrow$ indicates i, $\uparrow$ indicates min.

$$\left(\underset{\Uparrow}{12}, \underset{\uparrow}{5}, 8, 16, 9, 31\right)$$

$$\left(5, \underset{\Uparrow}{12}, \underset{\uparrow}{8}, 16, 9, 31\right)$$

$$\left(5, 8, \underset{\Uparrow}{12}, 16, \underset{\uparrow}{9}, 31\right)$$

Example

$\Uparrow$ indicates i, $\uparrow$ indicates min.

$$\left(\underset{\Uparrow}{12}, \underset{\uparrow}{5}, 8, 16, 9, 31\right)$$

$$\left(5, \underset{\Uparrow}{12}, \underset{\uparrow}{8}, 16, 9, 31\right)$$

$$\left(5, 8, \underset{\Uparrow}{12}, 16, \underset{\uparrow}{9}, 31\right)$$

$$\left(5, 8, 9, \underset{\Uparrow}{16}, \underset{\uparrow}{12}, 31\right)$$

Selection sort

Example

$\Uparrow$ indicates i, $\uparrow$ indicates min.

$$\left(\underset{\Uparrow}{12}, \underset{\uparrow}{5}, 8, 16, 9, 31\right) \qquad \left(5, 8, 9, \underset{\Uparrow}{16}, \underset{\uparrow}{12}, 31\right)$$

$$\left(5, \underset{\Uparrow}{12}, \underset{\uparrow}{8}, 16, 9, 31\right) \qquad \left(5, 8, 9, 12, \underset{\Uparrow}{16}, 31\right)$$

$$\left(5, 8, \underset{\Uparrow}{12}, 16, \underset{\uparrow}{9}, 31\right)$$

Selection sort

- Worst case time complexity: $O(n^2)$ (quadratic).
- Average case time complexity: $O(n^2)$.
- Space complexity: $O(1)$.
- In-place: Yes.
- Stable: No.

Example

The array $\{(2, A), (2, B), (1, C)\}$ will be sorted as $\{(1, C), (2, B), (2, A)\}$.

Bubble sort

Bubble sort sorts the array by repeatedly swapping non-ordered adjacent keys. After each for loop iteration, the maximum is moved (or *bubbled*) towards the end.

```
public static void bubbleSort(int A[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (A[j] > A[j + 1]) {
                // Swap A[j] with A[j + 1]
                int tmp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = tmp;
            }
        }
    }
}
```

Example

$$(12, 5, 8, 16, 9, |31)$$

Bubble sort

Example

$$(12, 5, 8, 16, 9, |31)$$
$$(5, 8, 12, 9, |16, 31)$$

Example

$$(12, 5, 8, 16, 9, |31)$$
$$(5, 8, 12, 9, |16, 31)$$
$$(5, 8, 9, |12, 16, 31)$$

Example

$$(12, 5, 8, 16, 9, |31)$$
$$(5, 8, 12, 9, |16, 31)$$
$$(5, 8, 9, |12, 16, 31)$$

$$(5, 8, |9, 12, 16, 31)$$

Example

$$(12, 5, 8, 16, 9, |31)$$
$$(5, 8, 12, 9, |16, 31)$$
$$(5, 8, 9, |12, 16, 31)$$
$$(5, 8, |9, 12, 16, 31)$$
$$(5, |8, 9, 12, 16, 31)$$

Bubble sort

- Worst case time complexity: $O(n^2)$ (quadratic).
- Average case time complexity: $O(n^2)$.
- Space complexity: $O(1)$.
- In-place: Yes.
- Stable: Yes.

Remark

Bubble sort performs a lot of swaps, and as a result it has in practice a poor performance compared to selection sort.

Merge sort

Merge sort is a divide-and-conquer algorithms to sort an array of n elements:

1. Divide the array into two equal parts.
2. Sort each part apart (recursively).
3. Merge the two sorted parts.

The key step in merge sort is merging two sorted arrays, which can be done in $O(n)$.

Example

Given two arrays $B = \{1, 4, 6\}$ and $C = \{2, 3, 7, 8\}$, the result of merging B and C is $\{1, 2, 3, 4, 6, 7, 8\}$.

```java
public static void mergeSort(int[] A, int l, int r) {
    if (l >= r)
        return;
    int m = (l + r) / 2;
    mergeSort(A, l, m); // Sort first half
    mergeSort(A, m + 1, r); // Sort second half
    merge(A, l, m, r); // Merge
}
```

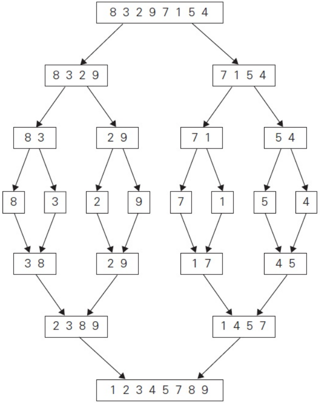
Merge sort

```
private static void merge(int[] A, int l, int m, int r) {
    int[] B = new int[r - l + 1];
    int i = l, j = m + 1, k = 0;
    while (i <= m && j <= r)
        if (A[i] <= A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    if (i > m)
        while (j <= r)
            B[k++] = A[j++];
    else
        while (i <= m)
            B[k++] = A[i++];
    for (k = 0; k < B.length; k++)
        A[k + l] = B[k];
}
```

King Saud University

Example

Sort the array: $8, 3, 2, 9, 7, 1, 5, 4.$



Merge sort

- Worst case time complexity: $O(n \log n)$ (sub-quadratic).
- Average case time complexity: $O(n \log n)$.
- Space complexity: $O(n)$ (requires auxiliary memory).
- In-place: No.
- Stable: Yes.