# Sorting

## CSC 212: Data Structures

King Saud University

## Outline

- The sorting problem.
- Properties of sorting algorithms.
- General purpose sorting algorithms:
    - Quadratic sorting algorithms: insertion sort, selection sort and bubble sort.
    - Sub-quadratic sorting algorithms: merge sort and quicksort.
- Specialized sorting algorithms: bucket sort, counting sort and radix 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*.

### Example

This is an instance of the sorting problem:

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

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

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

## 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

- The first type of sorting algorithms we are going to see are sorting 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).

## Insertion sort

Insertion sort gradually builds the sorted array by putting each new key in its correct position.

```
public static void insertionSort(int[] A, int n) {
    for (int i = 1; i < n; i++) {
        int j = i;
        while (j > 0 && A[j - 1] > A[j]) {
            int tmp = A[j];
            A[j] = A[j - 1];
            A[j - 1] = tmp;
            j--;
        }
    }
}
```

## Insertion sort

### Example

$\Downarrow$ indicates $i$, $\uparrow$ indicates $j$.

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

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

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

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

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

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

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

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

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

## Insertion 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.

## Selection sort

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

```
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;
    }
}
```

## Selection sort

### Example

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

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

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

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

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

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

## 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;
            }
        }
    }
}
```

## Bubble sort

### 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 insertion sort and 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 $\{1, 2, 3, 4, 6, 7, 8\}$.

## Merge sort

```
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
}
```

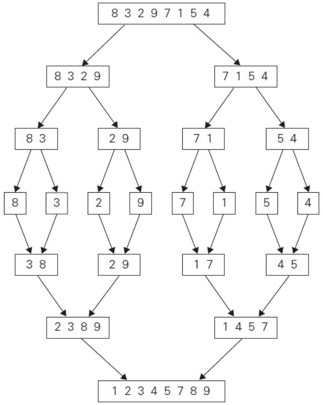
## Merge sort

```
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];
}
```

## Merge sort

### 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.

## Quick sort

Quick sort is another divide-and-conquer algorithms to sort an array of $n$ elements:

1. Pick any element of the array and call it the pivot (the first element, or a randomly chosen element for example) .
2. Rearrange the array so that all elements before the pivot are less or equal the pivot, and all those after the pivot are greater or equal to the pivot (**partitioning**).
3. Recursively sort the part of the array before the pivot and the one after the pivot.

## Quick sort

```
public static void quickSort(int[] A, int l, int r) {
    if (l < r) {
        int s = partition(A, l, r);
        quickSort(A, l, s - 1);
        quickSort(A, s + 1, r);
    }
}
```

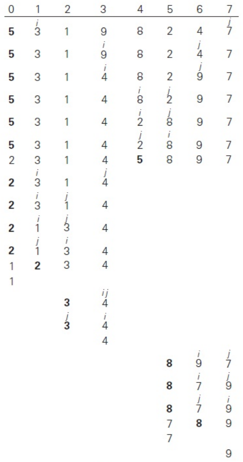
## Quick sort

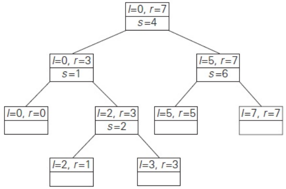
```
private static int partition(int[] A, int l, int r) {
   int p = A[l], i = l + 1, j = r;
   while (i < j) {
      while (A[i] <= p && i < j)
         i++;
      while (A[j] > p && i < j)
         j--;
      int tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
   }
   int s;
   if (A[i] <= p) s = i; else s = i - 1;
   int tmp = A[l];
   A[l] = A[s];
   A[s] = tmp;
   return s;
}
```

## Quick sort

### Example

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


(a)



(b)

## Quick sort

- Worst case time complexity: $O(n^2)$ (**quadratic**). The worst case happens when the array is already sorted for example.
- Average case time complexity: $O(n \log n)$ (**sub-quadratic**).
- Space complexity: $O(1)$.
- In-place: Yes.
- Stable: No.

**Example**

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

**Remark**

Although quick sort is quadratic in the worst case, in practice it is the fastest general purpose sorting algorithm. This is why it is the default sorting algorithm in virtually all standard libraries (C++, Java, etc.).

## Specialized sorting algorithms

- We have seen that the best comparison-base algorithms are $O(n \log n)$ worst case. Can we do better?
- The answer is **no**. It can be proved that: **no comparison-based algorithm can sort an array in less than $O(n \log n)$ in the worst case**.
- This result is for general data, but when the keys are of special type (like small positive integers or strings), we can find faster sorting algorithms.
- Specialized sorting algorithms do not use comparison in general and can only be used with specific types of keys.

## Bucket sort

Bucket sort is a non-comparison-based sorting algorithm that can be used to sort positive integer keys as follows:

1. Create an array of $k$ "buckets", initially all empty.
2. Put each element of $A$ in its bucket.
3. Sort each non-empty bucket (using insertion sort for example).
4. Concatenate all sorted buckets.

## Bucket sort

**Example**

Let $A = \{4, 25, 1, 18, 14, 29, 8, 7\}$, and $k = 3$

- Create an array of 3 empty buckets: Bucket 0 (for $0 \leq A[i] < 10$), Bucket 1 (for $10 \leq A[i] < 20$) and Bucket 2 (for $20 \leq A[i] < 30$).

| Bucket 0 | Bucket 1 | Bucket 2 |
|---|---|---|
| | | |

## Bucket sort

**Example**

Let $A = \{4, 25, 1, 18, 14, 29, 8, 7\}$, and $k = 3$

- Create an array of 3 empty buckets: Bucket 0 (for $0 \leq A[i] < 10$), Bucket 1 (for $10 \leq A[i] < 20$) and Bucket 2 (for $20 \leq A[i] < 30$).

| Bucket 0 | Bucket 1 | Bucket 2 |
|---|---|---|
| | | |

- Assign elements to buckets (we use a linked list to implement buckets):

| Bucket 0 | Bucket 1 | Bucket 2 |
|---|---|---|
| $4 \to 1 \to 8 \to 7$ | $18 \to 14$ | $25 \to 29$ |

## Bucket sort

### Example

- Sort buckets:

| Bucket 0 | Bucket 1 | Bucket 2 |
|---|---|---|
| $1 \rightarrow 4 \rightarrow 7 \rightarrow 8$ | $14 \rightarrow 18$ | $25 \rightarrow 29$ |

- Concatenate all sorted buckets back in $A$:

$$A : \{1, 4, 7, 8, 14, 18, 25, 29\}.$$

## Bucket sort

```
public static void bucketSort(int[] A, int n, int k) {
   // Create empty buckets
   List<Integer>[] buckets = new List[k];
   for (int b = 0; b < k; b++)
      buckets[b] = new LinkedList<Integer>();
   // Put each element in its bucket
   int max = max(A, n);
   max++;
   for (int i = 0; i < n; i++)
      buckets[k * A[i] / max].insert(A[i]);
   // Sort and concatenate buckets
   int i = 0;
   for (int b = 0; b < k; b++) {
      int[] B = sort(buckets[b]);
      for (int j = 0; j < B.length; j++)
         A[i++] = B[j];
   }
}
```

## Bucket sort

- Worst case time complexity: $O(n^2)$ (**quadratic**). The worst case happens when all keys fall in the same bucket.
- Average case time complexity: $O(n + \frac{n^2}{k} + k)$ (this becomes $O(n)$ when $k \approx n$).
- Space complexity: $O(n + k)$.
- In-place: No.
- Stable: Yes.

## Counting sort

Counting sort is a simple efficient sorting algorithm that does not use comparison. It is used when the keys are small positive integers.

## Counting sort

### Example

Consider the array $A = \{4, 5, 1, 4, 7, 3, 7, 0\}$. Since all values are less than 10, we create an array of size 10 that counts the frequency of each key in $A$ (any size $> max(A)$ will work):

1. Create an array called $counts$ and initialize it to 0: $counts = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$
2. Go through $A$ and increment $counts[A[i]]$: $counts = \{1, 1, 0, 1, 2, 1, 0, 2, 0, 0\}$
3. Compute the prefix sum of $counts$: $counts = \{1, 2, 2, 3, 5, 6, 6, 8, 8, 8\}$
4. Notice that $counts[A[i]] - 1$ contains the correct index of the last occurrence of $A[i]$. For example, $counts[A[3]] - 1 = 4$, which means the second 4 in $A$ should be in position 4. To get the position of the previous 4, we just decrement $counts[A[i]]$.
5. Create an array $B$ of the same size as $A$. Move through $A$ **backwards** and assign each key in $A$ to its correct position in $B$: $B[counts[A[i]] - 1] = A[i]$, then decrement $counts[A[i]] - -$: $B = \{0, 1, 3, 4, 4, 5, 7, 7\}$.
6. Finally, copy $B$ to $A$.

## Counting sort

```
public static void countingSort(int[] A, int n, int m) {
    int[] counts = new int[m];
    for (int j = 0; j < m; j++)
        counts[j] = 0;
    for (int i = 0; i < n; i++) // Count frequency
        counts[A[i]]++;
    for (int j = 1; j < m; j++) // Compute prefix sum
        counts[j] += counts[j - 1];
    int[] B = new int[n];
    for (int i = n - 1; i >= 0; i--) { // Put elements in correct order in B
        B[counts[A[i]] - 1] = A[i];
        counts[A[i]]--;
    }
    for (int i = 0; i < n; i++) // Copy back B to A
        A[i] = B[i];
}
```

## Counting sort

- Worst case time complexity: $O(n+m)$.
- Average case time complexity: $O(n+m)$.
- Space complexity: $O(n+m)$.
- In-place: No.
- Stable: Yes.

## Radix sort

- Radix sort is a non comparison-based sorting algorithm that can be used to sort positive integer keys.
- The algorithm considers the digits of the keys one by one and sorts the keys each time according to the selected digit.
- At each step the keys are sorted using simple sorting algorithm such as counting sort.
- Radix sort can also be used to sort strings.

## Radix sort

### Example

We want to sort the array $A = \{54, 875, 21, 4, 418, 313, 253, 540, 21, 74\}$. We are going to apply radix search by considering decimal digits at each iteration. Hence, our base $b = 10$. We can choose other bases: $b =$ 2, 4 or 256 for example.

1. Sort keys according to first digit:
   $\{4, 5, 1, 4, 8, 3, 3, 0, 1, 4\} \rightarrow \{540, 21, 21, 313, 253, 54, 4, 74, 875, 418\}$.
2. Sort keys according to second digit:
   $\{4, 2, 2, 1, 5, 5, 0, 7, 7, 1\} \rightarrow \{4, 313, 418, 21, 21, 540, 253, 54, 74, 875\}$.
3. Sort keys according to third digit:
   $\{0, 3, 4, 0, 0, 5, 2, 0, 0, 8\} \rightarrow \{4, 21, 21, 54, 74, 253, 313, 418, 540, 875\}$.

### Remark

How to extract digits?: 1st: $A[i]\%10$, 2nd: $(A[i]/10)\%10$, 3rd: $(A[i]/100)\%10$, etc.

## Radix sort

```
public static void radixSort(int[] A, int n, int b) {
    int[] B = new int[n];
    int dv = 1;
    while (true) {
        boolean done = true;
        for (int i = 0; i < n; i++) {
            B[i] = (A[i] / dv) % b; // Extract digit
            if (B[i] != 0) done = false;
        }
        if (done) break;
        int[] index = countingSortIndex(B, n, b);
        for (int i = 0; i < n; i++)
            B[index[i]] = A[i];
        for (int i = 0; i < n; i++)
            A[i] = B[i];
        dv *= b;
    }
}
```

## Radix sort

CountingSortIndex is the same as CountingSort, except that it returns the index instead of sorting $A$.

```
private static int[] countingSortIndex(int[] A, int n, int m) {
    int[] counts = new int[m];
    for (int j = 0; j < m; j++)
        counts[j] = 0;
    for (int i = 0; i < n; i++)
        counts[A[i]]++;
    for (int j = 1; j < m; j++)
        counts[j] += counts[j - 1];
    int[] index = new int[n];
    for (int i = n - 1; i >= 0; i--) {
        index[i] = counts[A[i]] - 1;
        counts[A[i]]--;
    }
    return index;
}
```

## Radix sort

Let $k$ be the maximum number of digits in any key.

- Worst case time complexity: $O(kn)$.
- Average case time complexity: $O(kn)$.
- Space complexity: $O(n + b)$.
- In-place: No.
- Stable: Yes.