# 8 Linear Time Sorting

In *comparison sorting*, the algorithm gains information about the order of items by comparing input elements two at a time. For example, in order to sort the input sequence $[a_0, a_1, \ldots, a_n]$ based on the value of a function *key*, the algorithm makes comparisons of the form $key(a_i) \leq key(a_j)$.

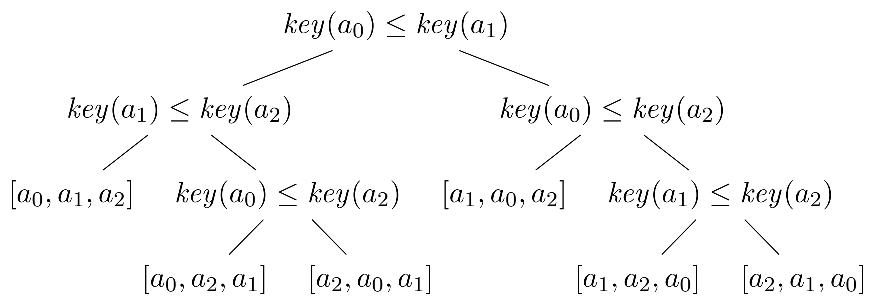
The worst-case running time of some commonly-used comparison sorting algorithms have an upper bound of $O(n^2)$ (e.g. for insertion sort and selection sort) or $O(n \log n)$ (e.g. for heap sort, merge sort, and quick sort).

## 8.1 Lower bound for the worst-case of comparison sorts

For the purpose of deriving a lower bound, without loss of generality, we can focus only on the number comparisons required to sort an array. A lower bound for the worst-case number of comparisons will also be a lower bound for the worst-case time complexity.

The collection of all the comparisons required for an algorithm to determine the correct order of elements can be represented as a full binary decision tree where the internal (non-leaf) nodes are comparisons and the leaf nodes are possible orderings. Different algorithms can produce different comparison decision trees.

**Example 8.1.** The following figure shows the comparison decision tree of the insertion sort algorithm when applied to a list of length 3 of the form $[a_0, a_1, a_2]$.



In every comparison decision tree:

- the complete execution of the algorithm on a given input can be seen as a path from the root of the tree to a leaf node;

- for each non-leaf node, if the result of the comparison is true, the left branch is taken; otherwise then the right branch is taken;
- the leaf nodes show how the items in the input should appear in the output; and
- there is a leaf node for every possible ordering of items.

The worst-case running time of a comparison sorting algorithm is proportional or greater than the worst-case number of comparisons made by the algorithm.

**Theorem 8.1.** *Any comparison sort algorithm requires* $\Omega(n \log n)$ *comparisons in the worst case.*

*Proof.* Let $n$, $l$, and $h$ respectively denote the size of input, the number of leaves in the tree, and the height of the tree. We view $h$ as a function of $n$.

- Number of possible outputs:
- Relation between $l$ and $n$:
- Relation between $l$ and $h$:
- Relation between $h$ and $n$:

$$\begin{aligned}
\log 2^h &\geq \log n! \\
h &\geq \log n! \\
&= \log n \times (n-1) \cdots \times 2 \times 1 \\
&= \underbrace{\log n + \log(n-1) + \cdots + \log 2 + \log 1}_{n \text{ terms}} \\
&\geq \underbrace{\log\lfloor\frac{n}{2}\rfloor + \cdots + \log\lfloor\frac{n}{2}\rfloor}_{\lfloor\frac{n}{2}\rfloor \text{ terms}} \\
&= \lfloor\frac{n}{2}\rfloor \log\lfloor\frac{n}{2}\rfloor \\
&\in \Theta(n \log n)
\end{aligned}$$

Thus $h \in \Omega(n \log n)$.

□

## 8.2 Non-comparison sorting

The lower bound in the above theorem applies to problems where the key values must be compared. If the key function returns a natural number from a bounded range, better running times can be achieved. In the following discussion and algorithms we use $key(a_i)$ to determine the position of $a_i$ in the output array without making any comparison.

**Question 8.1.** You have an array $A[0\,..\,n]$ of objects. You are to sort the array based on a key with values between 0 and $n$ and is guaranteed to be distinct (that is, no two objects will have the same key value). Can this be achieved in linear time?

Example:

| *object* | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ |
|---|---|---|---|---|---|---|
| *key*(*object*) | 2 | 4 | 3 | 1 | 5 | 0 |

**Question 8.2.** You have an array $A[0\,..\,n]$ of objects. You are to sort the array based on a key whose value is between 0 and $k$ and is guaranteed to be distinct and we have $k \geq n$. Can this be achieved in linear time?

Example:

| *object* | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|---|
| *key*(*object*) | 0 | 3 | 6 | 4 | 1 |

**Question 8.3.** You have an array $A[0\,..\,n]$ of objects. You are to sort the array based on a key whose value is between 0 and $k$ with $k \geq n$. Can this be achieved in linear time?

Example:

| $object$ | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|---|
| $key(object)$ | 6 | 1 | 6 | 2 | 1 |

## 8.3 Counting sort

Counting sort assumes that each of the $n$ input elements has a key value in the range 0 to $k$, for some natural number $k$. The algorithm has two stages:

1. The algorithm calls the procedure Key-Positions which counts the number of times each key value occurs in the input array and uses this information to compute the position of objects with that key in the output (sorted) array. It returns an array of positions of length $k+1$.
2. The algorithm uses the positions obtained in the first stage to place the elements of the input at the right position in the output array.

```
1  procedure Counting-Sort($A, key$)
2    create an output array $B$ with the same size as $A$
3    $P \leftarrow$ Key-Positions($A, key$)
4    for $a$ in $A$
5      $B[P[key(a)]] \leftarrow a$
6      $P[key(a)] \leftarrow P[key(a)] + 1$
7    return $B$
```

The input to the algorithm is an array $A$ of objects and a key function according to which a sorted output array must be generated. After line 3, $P[i]$ contains the starting index for objects whose key value is $i$. In other words, the first object in $A$ whose key value is $i$ must be placed at index $P[i]$ in the output array. The for-loop goes through the elements of $A$ and places each element in its correct sorted position in the output array $B$, and updates $P$.

```
1  procedure Key-Positions($A, key$)
2    $k \leftarrow \max\{key(a) : a \in A\}$
3    let $C[0\,..\,k]$ be a new array
4    for $i$ from 0 to $k$
5      $C[i] \leftarrow 0$
6    for $a$ in $A$
7      $C[key(a)] \leftarrow C[key(a)] + 1$
8    $sum \leftarrow 0$
9    for $i$ from 0 to $k$
10     $C[i], sum \leftarrow sum, sum + C[i]$
11   return $C$
```

At line 2 the algorithm finds the maximum value of the key function over the input array $A$ and stores it in $k$. At lines 3–5, a new array $C$ of size $k+1$ is created for keeping counts; it is initialised with zeros.

After the for-loop at lines 6–7, $C[i]$ is the number of elements whose key value is $i$. The for-loop at lines 9–10 computes a running sum over $C$. At the end of the for-loop, $C[i]$ is the number of elements whose key value is less than $i$.

**Example 8.2.** Trace KEY-POSITIONS and COUNTING-SORT on the following input.

| *object* | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ |
|---|---|---|---|---|---|---|---|---|
| *key*(*object*) | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

Value of $C$ after line 7:

Value of $C$ after line 10:

Values of $B$ and $P$ during the execution of COUNTING-SORT:

## Analysis and properties of counting sort

Let $n$ be the size of the input array. The time complexity of KEY-POSITIONS is as follows. Line 2 takes $\Theta(n)$ time. The three for-loops take $\Theta(k)$, $\Theta(n)$, and $\Theta(k)$ time respectively.

In the main body of the counting sort algorithm, the for-loop takes $\Theta(n)$ time. Therefore the time complexity of the counting sort algorithm is $\Theta(k + n) = \Theta(\max(k, n))$. This means that if $k = O(n)$, the algorithm runs in $\Theta(n)$ time.

An important property of counting sort is that it is **stable**: objects with the same key value appear in the output array in the same order as they do in the input array. This is equivalent to saying that the algorithm breaks ties between two objects with the same key value by the rule that whichever object appears first in the input array appears first in the output array.

**Example 8.3.** Let $A = [4, 2, 4, 3, 2, 3]$ and let the key be the identity function. What is the intermediate state and output of KEY-POSITIONS on this input?

- The array containing key value counts is: $[0, 0, 2, 2, 2]$.
- The output is $[0, 0, 0, 2, 4]$.

**Example 8.4.** Let $A = [a, b, c, d, e, f, g, h]$ where each element is an object and the key values are:

| *object* | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* |
|---|---|---|---|---|---|---|---|---|
| *key*(*object*) | 2 | 4 | 5 | 0 | 2 | 4 | 3 | 5 |

1. What is the output of KEY-POSITIONS? Show one intermediate step.
2. Trace the procedure COUNTING-SORT using the key positions from the previous step.

1. The key counts (in the intermediate step) are $[1, 0, 2, 1, 2, 2]$. The output of KEY-POSITIONS is $[0, 1, 1, 3, 4, 6]$.

2.

| *P* | *Sorted array* |
|---|---|
| (0, 1, 1, 3, 4, 6) | (-, -, -, -, -, -, -, -) |
| (0, 1, 2, 3, 4, 6) | (-, a, -, -, -, -, -, -) |
| (0, 1, 2, 3, 5, 6) | (-, a, -, -, b, -, -, -) |
| (0, 1, 2, 3, 5, 7) | (-, a, -, -, b, -, c, -) |
| (1, 1, 2, 3, 5, 7) | (d, a, -, -, b, -, c, -) |
| (1, 1, 3, 3, 5, 7) | (d, a, e, -, b, -, c, -) |
| (1, 1, 3, 3, 6, 7) | (d, a, e, -, b, f, c, -) |
| (1, 1, 3, 4, 6, 7) | (d, a, e, g, b, f, c, -) |
| (1, 1, 3, 4, 6, 8) | (d, a, e, g, b, f, c, h) |

## 8.4 Radix sort

In counting sort if $k$ becomes larger than $n$, the algorithm loses its linear time property. So how to sort an array of large numbers?

The idea of radix sort is to sort numbers digit by digit. Counterintuitively, radix sort starts sorting numbers first based on their *least significant* (rightmost) digit, then sorts the result based on the second-least significant digit, and so on. If the numbers in the input array have at most $d$ digits, then after $d$ iterations the resulting array is sorted.

```
procedure Radix-Sort($A, d$)
  for $i$ from 1 to $d$
    use a stable sort to sort array $A$ on digit $i$
```

Radix sort requires another sorting algorithm that is stable. Counting sort is usually used for this purpose.

**Example 8.5.** Let $A = [329, 457, 657, 839, 436, 720, 355]$. Sort the array using radix sort. Show the result of each iteration.

```
Iteration 1: [720, 355, 436, 457, 657, 329, 839]
Iteration 2: [720, 329, 436, 839, 355, 457, 657]
Iteration 3: [329, 355, 436, 457, 657, 720, 839]
```

**Example 8.6.** Let $A = [543, 774, 166, 298, 7, 54, 12]$. Trace the procedure call Radix-Sort($A$, 2).

```
Iteration 1: [12, 543, 774, 54, 166, 7, 298]
Iteration 2: [7, 12, 543, 54, 166, 774, 298]
```