8 Linear Time Sorting

So far you have seen several sorting algorithms. Some of these algorithms have an upper bound of $O(n^2)$ for their running time (for example, insertion sort and selection sort) and some have an upper bound of $O(n \log n)$ (for example, heap sort, merge sort and quick sort).

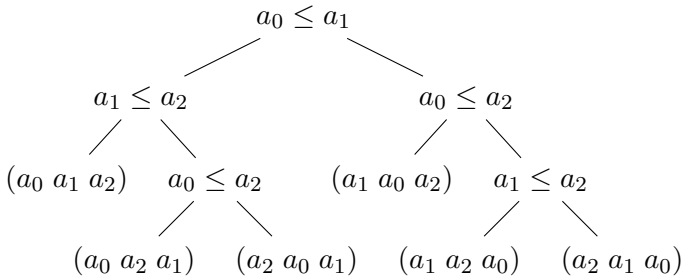
These algorithms share a common property: they *compare* the key of items to be sorted. For this reason we call them **comparison sort** algorithms.

8.1 Lower bound for comparison sorts

In comparison sorting, the algorithm gains information about the order of items by comparing input elements two at a time. For example, for an input sequence of $a_0, a_1, \ldots, a_n$, the algorithm makes comparisons of the form $a_i \leq a_j$ to determine which one comes earlier.

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.

Example 8.1. Draw the comparison decision tree of the sequence $[a_0, a_1, a_2]$.



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

Proof. Assume n distinct elements:

- number of possible outputs:
- minimum number of leaf nodes:
- minimum height of the tree:
- worst-case time (maximum path length):

8.2 How about 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!

For the following it helps to distinguish between the objects in the input and their key values, so comparisons take the form $key(a_i) \leq key(a_j)$.

Question 8.1. 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 n and is guaranteed to be distinct (that is, no two objects will have the same key value). Give a linear time algorithm to achieve this.

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 $k \geq n$. Give a linear time algorithm to achieve this.

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 $k \geq n$. Give a linear time algorithm to achieve this.

Example:

$object$	a_0	a_1	a_2	a_3	a_4
$key(object)$	6	1	6	1	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 ← KEY-POSITIONS(A, key)
4    for a in A
5      B[P[key(a)]] ← a
6      P[key(a)] ← 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 ← max{key(a) : a ∈ A}
3    let C[0..k] be a new array
4    for i from 0 to k
5      C[i] ← 0
6    for a in A
7      C[key(a)] ← C[key(a)] + 1
8    sum ← 0
9    for i from 0 to k
10     C[i], sum ← 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 during lines 6 and 7:

Value of C during lines 9 and 10:

Values of B and P during the execution of COUNTING-SORT.

8.3.1 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.

```
1 procedure RADIX-SORT(A, d)
2   for i from 1 to d
3     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. ...

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: $[7, 12, 543, 54, 166, 774, 298]$.

9 Reductions and hardness of problems

Reduction is the act of *reducing* one problem to another. Problem X can be reduced to problem Y if and only if every instance of X can be turned into an instance of Y and the solution of this instance of Y can be converted back into a solution for X.

Example 9.1. The problem of finding the minimum of an array of numbers can be reduced to the problem of finding the maximum element in the array.

```
def find_min_using_max(numbers):
    negated_numbers = [-x for x in numbers]
    maximum = max(negated_numbers)
    minimum = -maximum
    return minimum
```

Answer the following questions:

- What is the time complexity of this algorithm?
- What can be said about the difficulty of the problem of finding the min?
- What can be said about the difficulty of the problem of finding the max, if we know that the time complexity of *any* algorithm for finding the min is in $\Omega(n)$?

More generally, a computational problem X can be reduced to a computational problem Y if there is a procedure such as the following:

```
1 procedure SOLVE-X-USING-Y(X-instance)
2   Y-instance ← CONVERT-INSTANCE-X-TO-Y(X-instance)
3   Y-solution ← SOLVE-Y(Y-instance)
4   X-solution ← CONVERT-SOLUTION-Y-TO-X(Y-solution)
5   return X-solution
```

Reduction of X to Y allows two types of conclusions:

- if Y is easy, then X is easy.
- if X is hard, then Y is hard.

Y is said to be easy if an efficient algorithm for solving it is known.

X is said to be hard if it has been proved that no efficient algorithm for solving X can exist or no efficient algorithm for solving X is known.

More precisely,

- if the time complexity of conversion (of both instance and solution) is in $O(f_{convert})$, and the time complexity of solving Y is in $O(f_{solveY})$, then the time complexity of X is in $O(f_{solveY} + f_{convert})$;
- if we know that any algorithm for X is in $\Omega(f_{solveX})$, then any algorithm for solving Y is in $\Omega(f_{solveX} - f_{convert})$.

Example 9.2. The problem of finding the minimum of an array of numbers can be reduced to the problem of sorting the array.

```
def find_min_using_sort(list_of_numbers):
    sorted_list = comparison_sort(list_of_numbers)
    return sorted_list[0]
```

Answer the following questions:

- What can be said about the worst-case running time of this function without making any assumption about the sorting algorithm (other than it is a comparison sort)? [The two rules mentioned above are not needed to answer this question.]
- If we can prove that all algorithms for finding min are in $\Omega(n)$, what can be said about the problem of sorting?
- Knowing that there are sorting algorithms that run in $O(n \log n)$, what can be said about the problem of finding the min.
- If we devise a separate algorithm that can solve the problem of finding the min in $O(n)$, what can be said about the sorting problem (from the above reduction)?
- Considering the fact that the worst-case running time of a comparison sorting algorithm is in $\Omega(n \log n)$, what can be said about the difficulty of the min problem?

Example 9.3. Reduce the problem of sorting an array of numbers to the topological sorting problem. What claims can be made for the complexity of the two problems?