Chapter 8
Sort in Linear Time

We have so far discussed several sorting algorithms that sort a list of \( n \) numbers in \( O(n \log n) \) time. Both the space hungry mergesort and the structurely interesting heapsort achieve this upper bound in the worst case, while the quick sort algorithm does it on average.

For each of these algorithms, we can produce a sequence that will demonstrate these bounds.

These algorithms, together with some of the previously discussed ones, such as insertion sort and selection sort, are all \textit{comparison based}.

We will show first that, for all such comparison based algorithms, it is the best they could do 😊.
Complexity bounds

The function $f(n)$ is an upper bound for a problem, iff at least one algorithm exists that solves the problem in $O(f(n))$.

One way to establish this upper bound for a problem is to develop such an algorithm in $O(f(n))$. We certainly hope that we can reduce the upper bound of a problem as much as we can. This topic is called algorithmics.

On the other hand, $g(n)$ is a lower bound of a problem, iff every algorithm, including all the existing ones, and those that might be found out later, for this problem is $\Omega(g(n))$.

This $g(n)$ is usually difficult to identify 😞. We will further explore this issue in the complexity theory chapter, or in CS 3780.

But, sometimes, it is “easy” to establish a lower bound for a class of problems ☺.
We are lucky with sorting...

It is easy to see that it takes $\Omega(n)$ to sort a list of $n$ numbers, since it takes that much time to look at those numbers to ensure that they will all be put in the right place.

Also, it takes $\Omega(n^2)$ to multiply two $n \times n$ matrices, as the result contains $n^2$ elements, each of which must be generated in $\Omega(1)$ time. (It actually takes $\Theta(n^3)$ as we saw earlier.)

We now dig out the complexity of the sorting problem, i.e., how many comparisons are needed to sort a list of $n$ elements, by following the comparison based approach?

It turns out to be $\Omega(n \log n)$. 
Decision tree

A decision tree is a binary tree that represents the comparisons between elements made by a sorting algorithm. Each and every one of its internal nodes represents a set of possible ordering, consistent with comparisons made so far. In particular, each leaf represents a permutation of the original input data.

Below is an example of a decision tree:

Decision tree and sorting

The execution of any sorting algorithm essentially traces a path in an associated decision tree, from the top to a leaf. At each internal node $[i : j]$, a comparison $A[i] \leq A[j]$ is made. If the result is positive, the algorithm continues to trace the left subtree; otherwise, it goes to the right subtree.

When it eventually reaches a leaf, it has established the ordering of all the elements in the original array.

**Question:** Which sorting mechanism leads to the previous tree?

**Answer:** Insertion sort (?).

**Homework:** Construct a decision tree for the selection sort algorithm when applied to three elements, $(A[1], A[2], A[3])$. 
Why decision trees?

Since any correct sorting algorithm has to produce each and every $n!$ permutation of the input items, a necessary condition is that each of the $n!$ permutations must show up as one of the leaves of the associated decision tree.

All of such leaves must be reachable from the root of this tree by a path corresponding to an actual execution of the sorting algorithm.

Thus, (average) number of comparisons made in a sorting algorithm is equal to the (average) height of the corresponding binary decision tree.

Notice that such a decision tree must contain at least $n!$ leaves, thus has to be “tall” 😊.
Basic results

**Lemma 1** Let $T$ be a binary tree of height $h$, then $T$ has at most $2^h$ leaves.

**Proof by induction on $h$:** The result is trivial when $h = 0$ since the tree with its height being 0 contains exactly one node.

Assume that it is true for any tree of height strictly less than $h$. Let $T$ be a tree of height $h > 0$, then the root can’t be a leaf and both left and right subtrees have heights at most $h-1$, thus, by inductive assumption they both have at most $2^{h-1}$ leaves. As a result, $T$ has at most $2^h$ leaves. □

**Lemma 2** A binary tree with $L$ leaves must have height at least $\lceil \log(L) \rceil$.

**Proof:** Let $h$ be its height, by the previous lemma, $L \leq 2^h$, or $h \geq \log_2 L$.

Since $h$ is an integer, $h \geq \lfloor \log(L) \rfloor$. □
Theorem 1 Any comparison based sorting algorithm requires at least $\lceil log(n!) \rceil$ comparisons as its worst case.

Proof: For a list of $n$ numbers, its decision tree has at least $L (\geq n!)$ leaves, since each of the $n!$ permutations must be a leaf of the associated decision tree. The result follows from Lemma 2. □

Theorem 2 Any comparison based sorting algorithm requires $\Omega(n log(n))$ comparisons.

Proof: We have that

$$log(n!) = log(n \cdot (n-1) \cdot \cdot \cdot \cdot 2 \cdot 1)$$
$$= log(n) + \cdot \cdot + log(1)$$
$$\geq log(n) + \cdot \cdot + log(\frac{n}{2})$$
$$\geq log(\frac{n}{2}) + \cdot \cdot + log(\frac{n}{2})$$
$$= \frac{n}{2} log(\frac{n}{2}) = \frac{n}{2} (log n - 1).$$

Result now follows from definition. □
Now what?

**Corollary 8.2:** Heapsort and Mergesort are asymptotically optimal comparison based sorts.

**Proof:** The $O(n \log n)$ upper bound on the running time for both Heapsort and Mergesort match the $\Omega(n \log n)$ lowbounds as proved in Theorem 1 for any comparison based sorting algorithms.

**Homework:** Exercises 8.1-1.

We will now explore sorting algorithms that are *not* comparison based, thus not subject to the previous restriction. In other words, it could be better. 😊
Counting sort

When discussing the *counting sort*, we assume that each of the $n$ elements is an integer in the range 0 to $k$, for some positive integer $k$.

When $k = O(n)$, counting sort runs in $\Theta(n)$ time.

The basic idea of this algorithm is to determine, for each input element $x$, the number of elements that are less than or equal to $x$. This information tells us where $x$ should be placed.

For example, if we know that 5 elements are less than or equal to $A[3]$, then it should be placed in position 5 in the sorted list.

And if we know this number for $(A[1], A[2], A[3], A[4], A[5])$ are $(2, 1, 5, 4, 3)$, then, the sorted form must be $(A[2], A[1], A[5], A[4], A[3])$.  


The code

In the following code, we assume that the inputs are put in an array $A[1..n]$.

COUNT-SORT($A$, $B$, $k$)
1. for $i<-0$ to $k$  // $k$ is the max value in $A$
2. do $C[i]<-0$
3. for $j<-1$ to length[$A$]
4. do $C[A[j]]<-C[A[j]]+1$
5. // $C[A[j]]$ now contains the number that $A[j]$
   // occurs in $A$. See Part (a) in the next page
6. for $i<-1$ to $k$
7. do $C[i]<-C[i]+C[i-1]$
8. // $C[i]$ now contains the number of elements
    // less than or equal to $A[i]$. See (b)
9. for $j<-\text{length}[A]$ downto 1  // Why backwards?
13. // Where should next occurrence of $A[j]$
    // be placed?

Let’s look at an example... .
An example

Below gives a concrete example of counting sort.

Here Part (a) shows the original data and the work done at the end of line 4; Part (b) reflects the work done in lines 6-7, Parts (c), (d) and (e) show what happen after one, two and three items are put in; and Part (f) shows the final result.
A bit explanation

Lines 1-4 collects the information, for each element $A[j]$, how many elements exist in the list that are less than or equal to $A[j]$.

If all the elements are distinct, the accessory array $C$ gives exactly where all the input elements should be placed in the $A$ list, since in a sorted list, an element is placed in position $j (\geq 0)$, iff there are precisely $j$ elements that are less than or equal to this element. This is what line 10 does.

To allow the possibility that $A[j]$ occurs multiple times, we use line 11 to decrement this position for $A[j]$ by 1, thus when the next occurrence of $A[j]$ comes in, it will be placed in the position to the left of the position where the previous copy was placed.

This is what we refer to as the stable property.
Running time

Both lines 1-2, and lines 6-7 take $\Theta(k)$, while both lines 3-4 and lines 9-11 take $\Theta(n)$. Thus, the overall time is $\Theta(k+n)(= \max\{k, n\})$. (Cf. Exercise 3.1-1)

When $k = O(n)$, the running time of the counting sort is $\Theta(n)$.

Although not “in-place”, Counting sort beats the previously discussed sorting algorithms in terms of running time, since it is not a comparison based sort.

The order of the last loop makes counting sort a stable sort, elements with the same value occur in the output in the same order as they do in the input.

This is particularly important for the counting sort since it is often used in the radix sort as a subroutine.

Homework: Exercises 8.2-1 and 8.2-3.
Radix sort

Given a list of numbers consisting of digits, in base 10, the *radix sort* sorts the numbers on their least significant digits first. The numbers whose least significant digits is $i \in [0, 9]$ is collectively in bin $i$. The numbers of bin 0, bin 1, ..., bin 9 are then collected as a list again. The radix sort then sorts on the whole list on the second-least significant digit, and put number into bins according to that digit.

The important thing is that if we always use a stable algorithm to distribute the numbers into the bins, then after this process, they are not only sorted on their second-least significant digit, but they also stay sorted on their least significant digit.

The process continues until they are sorted by their most significant digit. Again, if we always use a stable algorithm for all the steps, then at the end, the list is truly sorted.
An example

Below gives a concrete example of radix sort.

\begin{itemize}
  \item[(a)] Input chain
  \begin{align*}
  216 & \rightarrow 521 & \rightarrow 425 & \rightarrow 116 & \rightarrow 91 & \rightarrow 515 & \rightarrow 124 & \rightarrow 34 & \rightarrow 96 & \rightarrow 24
  \end{align*}
  \item[(b)] Chain after sorting on least significant digit
  \begin{align*}
  521 & \rightarrow 91 & \rightarrow 124 & \rightarrow 34 & \rightarrow 24 & \rightarrow 425 & \rightarrow 515 & \rightarrow 216 & \rightarrow 116 & \rightarrow 96
  \end{align*}
  \item[(c)] Chain after sorting on 2nd least significant digit
  \begin{align*}
  515 & \rightarrow 216 & \rightarrow 116 & \rightarrow 521 & \rightarrow 124 & \rightarrow 24 & \rightarrow 425 & \rightarrow 34 & \rightarrow 91 & \rightarrow 96
  \end{align*}
  \item[(d)] Chain after sorting on most significant digit
  \begin{align*}
  24 & \rightarrow 34 & \rightarrow 91 & \rightarrow 96 & \rightarrow 116 & \rightarrow 124 & \rightarrow 216 & \rightarrow 425 & \rightarrow 515 & \rightarrow 521
  \end{align*}
\end{itemize}

**Homework:** Exercises 8.3-1 and 8.3-2.
The code

RADIX-SORT(A, d)
1. for i ← d to 1
2. Use a stable sort to sort A on digit i

Lemma 8.3. Given $n$ $d$–digit numbers in which each digit can take on up to $k$ possible values, Radix-sort correctly sorts all these numbers in $\Theta(d(n + k))$ time.

Proof: Correctness can be argued with an inductive proof. Regarding the running time, when each digit is in the range 0 to $k - 1$, and $k$ is not too large, counting sort can be used as the intermediate step. Each pass of sorting $n$ $d$–digit numbers take time $\Theta(n + k)$, and there are $d$ passes.

Particularly, when $d$ is a constant, and $k = O(n)$, radix sort takes only linear time.