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 HeapSort, based on the interesting heap structure, achieve this upper bound in the worst case, while the QuickSort algorithm does it on average. For each of these algorithms, we can produce a sequence that will demonstrate these bounds.

These algorithms, and BubbleSort, InsertionSort and SelectionSort, are all comparison based as the core operation is to compare two piece of data.

Question: Can we do even better?

Answer: 1) For all such comparison based algorithms, $\Omega(n \log n)$ is the best they could do ☹. 2) There are $O(n)$ sorting algorithms that are not based on key comparisons ☺.
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 Computational Theory.

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 at least that much time to look at those numbers to ensure that they will all be put in the right place.

It takes at most $O(n^2)$ time to do comparison based sorting, since we have got a few such algorithms, including InsertionSort.

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 on Page 11 of the opening Chapter notes.)

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

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

What can a decision tree do?

Given three numbers, \((a_1, a_2, a_3)\), at the very beginning, we only know that all the six permutations could be the final output.

When we go down the tree, we will gradually cut down the possibilities. When we hit a leaf, we know exactly what the order of the original numbers is.
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 turns 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:** *InsertionSort*. You need to go back to the Insertion Sort algorithm (Page 5 of the *Basic analysis* chapter), and *trace through* this sort on the list $(a_1, a_2, a_3)$ to get the whole tree.

**Question:** Should we trust Dr. Shen? 😃
The right half of the tree...

**Homework:** Construct a decision tree for the Selection Sort algorithm when applied to three elements, \((a_1, a_2, a_3)\).
What do they look like?

Since any correct sorting algorithm has to produce each and every one of $n!$ 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. Recall that the height of a tree is the length of the longest path from the root to a leaf. (Cf. Page 5 of the HeapSort notes)

Notice that such a decision tree must contain at least $n!$ leaves, thus has to be “tall” 😞.
Are you kidding me?

We know that the best, worst, and average case for insertion sorting a list of \( n \) items is \( n - 1, \frac{n(n - 1)}{2}, \) (Cf. Page 28, Chapter 2 notes); and \( \frac{n(n+3)}{4} - 2H_n + 1 \) (Cf. Page 11, Chapter 5 notes).

If you look at the decision tree for the insertion sort, the length of the shortest path, and the longest path, is 2 and 3, matching the above results.

The average height of the decision tree for \( n = 3 \) case is

\[
\frac{2 \times 2 + 3 \times 4}{6} = \frac{16}{6} = \frac{8}{3} = 2\frac{1}{3}.
\]

The theoretical result gives \( 2\frac{1}{2} \).

Pretty close... 😊
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, $h \geq \lceil \log(L) \rceil$. □
Here we go...

**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
\[
\log(n!) = \log(n \cdot (n - 1) \cdots 2 \cdot 1)
\]
\[
= \log(n) + \cdots + \log(1)
\]
\[
\geq \log(n) + \cdots + \log\left(\frac{n}{2}\right)
\]
\[
\geq \log\left(\frac{n}{2}\right) + \cdots + \log\left(\frac{n}{2}\right)
\]
\[
= \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2} (\log n - 1).
\]
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 $\Omega(n \log n)$ lower bound as proved in Theorem 1 for any comparison based sorting algorithms.

**Homework:** Exercises 8.1-1.

If it is comparison based, it takes $\Omega(n \log n)$. Thus, *if we want something better, it cannot be comparison based.*

$$A \rightarrow B \equiv \neg B \rightarrow \neg A.$$  

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 😊 if we give up comparisons.
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 $k$.

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, i.e., “$\leq$”, $x$, thus allowing duplicates. This information tells us where $x$ should be placed.

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


For example, $(15, 13, 58, 40, 32)$ goes to $(13, 15, 32, 40, 58)$.

**Question:** How to dig out this information?
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 //Initialize $C$, the counters
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. //occurs in $A$. See Part (a) in the next page
7. for $i$<-1 to $k$
8. do $C[i]$<-$C[i]$+$C[i-1]$
9. // $C[i]$ now contains the number of elements
10. //less than or equal to $A[i]$, thus the position
11. //of $A[i]$ in the sorted list. See (b)
12. for $j$<-length[$A$] downto 1 //Why backwards?
15. //Where should next occurrence of $A[j]$ be placed? See Part c through f

Question: What I do if I feel dizzy... . 😞

Answer: Trace it through... . 😊
Here Part (a) shows the input $A$, and $C$ collects the work done at the end of line 4, where $k = 5$.

Part (b) reflects the work done in lines 7-8. $C$ now keeps the counts of all in $A$.

Parts (c), (d) and (e) show what happen after one, two and three items are put in $B$.

Part (f) shows the final result as kept in $B$. 
A demo... .

After going through Lines 1 and 2, for each $A[j]$, $C[A[j]]$ holds the number of occurrences of $A[j]$ in $A$. For example, $C[0]=2$ since there are two 0’s in $A$.

After going through Lines 7 and 8, $C[i]$ contains the number of items no bigger than $i$. For example, $C[0]=2$ means that there are two items in $A$ no bigger than 0; and $C[2]=4$ means that there are at most 4 items no bigger than 2, i.e., 0, 0, 2, 2. Finally, there are at most 8 items no bigger than 5, the maximum in $A$.

We now go from right to left in $A$. Where should $A[8]=3$ be placed? Since $C[3]=7$, there are 7 items in $A$ no bigger than 3, so 3 should be placed in position 7 in $B$. Since there could be other occurrence of 3, we decrement $C[3]$ by 1 to 6, as shown in Part (c).

We repeat this process in Lines 12 through 14 for every item in $A$, as shown in Part (f).
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 13 to decrement this position for $A[j]$ by 1, thus when the next occurrence of $A[j]$, which is the one to the left of the previous one, 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 7-8 take $\Theta(k)$, while both lines 3-4 and lines 12-14 take $\Theta(n)$. Thus, the overall time is $\Theta(k+n) (= \max\{k, n\})$. (Cf. Exercise 3.1-1)

When $k = O(n)$, i.e., $k = cn$, the running time of the counting sort is $\Theta(n)$. Although not “in-place” (Cf. Page 1, HeapSort notes), Counting sort beats the previously discussed sorting algorithms in terms of running time. Notice that it is not a comparison based sort.

Notice that the decreasing 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? I want to see...

Given the following list of numbers:

216, 521, 425, 116, 91, 515, 124, 34, 96, 24

We *stable sort* them (how?) with the *right-most digit*, and collect the following after the first pass.

521, 91; 124, 34, 24; 425, 515; 216, 116, 96;

We then sort all with the *middle digit*, notice that, since we follow a stable sort, for those who share the middle digit, they are sorted in the right most digit. 😊

515, 216, 116; 521, 124, 24, 425; 34; 91, 96;

**Question:** What data structure should we use here?

**Answer:** A list of FIFO queues
Finally, we sort them by the *leftmost digit*, or the most significant digit.

Notice that 21 is really 021. Moreover, they are not only sort by the leftmost digit. Since it is a stable sort, when they share the leftmost digit, they are already sorted by the latter two.

For example, with 116, and 124, they are in good order. 😊

24, 34, 91, 96; 116, 124; 216; 425; 515, 521

*You get to play with it, with some other numbers, until you fully understand this process.*

**Homework:** Think about 8.3-3.
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, while reading them through *from left to right*. The numbers whose least significant digits is \( i \in [0, 9] \) is collectively in a bucket, actually a queue, \( i \).

Those bucket 0, bucket 1, ..., bucket 9 are then collected, *bottom up*, as a list again. The whole list will be sorted on the second-least significant digit, and put number into bins according to that digit.

The important thing is that the above order ensures a stable sorting on the least significant digit. Thus, after this process for *the next bit*, 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, and we are done.
The code

RADIX-SORT(A, d)
1. for i←d to 1 // To ensure a stable sort
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. (Exercise 8.3-3) Regarding the running time, when each digit is in the range 0 to $k - 1$, and $k$ is not too large, i.e., $k = O(n)$, 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. 😊
# A summary

Below are the eight sorting algorithms that we have gone through, and the main results.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparison</th>
<th>Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$\frac{n^2}{2}$ (Project 3)</td>
<td>Best: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Avg.: $\frac{3n^2}{4}$ (Project 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Worst: $\frac{n^2}{2}$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Best: $n$ (P .28, Ch .2)</td>
<td>Best: $\frac{n^2}{2}$ (P .29, Ch .2)</td>
</tr>
<tr>
<td></td>
<td>Avg.: $\frac{n^2}{4}$ (P .11, Ch .5)</td>
<td>Avg.: $\frac{n^2}{4}$ (P .11, Ch .5)</td>
</tr>
<tr>
<td></td>
<td>Worst : $\frac{n^2}{2}$ (P .28, Ch .2)</td>
<td>Worst: $\frac{n^2}{2}$ (P .29, Ch .2)</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\frac{n^2}{2}$ (Project 3)</td>
<td>3n (Project 3)</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log n$ (P .26, Ch .6)</td>
<td>$n \log n$ Homework 6.4.-3.</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log(n)$ (P .55, Ch .2)</td>
<td>$n \log(n)$ (Project 3)</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Best: $n \log n$ (P .12, Ch .7)</td>
<td>Best: $n \log n$</td>
</tr>
<tr>
<td></td>
<td>Avg.: $n \log n$ (P .21, Ch .7)</td>
<td>Avg.: $n \log n$</td>
</tr>
<tr>
<td></td>
<td>Worst : $\frac{n^2}{2}$ (P .13, Ch .7)</td>
<td>Worst : $\frac{n^2}{2}$</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$\Theta(k + n)$ (P .18, Ch .8)</td>
<td>$3n + k$ (P .14, Ch .8)</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$\Theta(d(k + n))$ (P .22, Ch .8)</td>
<td>$(3n + k)d$ (P .22, Ch .8)</td>
</tr>
</tbody>
</table>

Here we only write down what and where the why and how are discussed in the notes. For example, the average number of movements for the BubbleSort is obtained through a practical analysis in the sampler to Project 3; while the best time of the InsertionSort is derived on Page 28 in the Chapter 2 notes.