In COMS12100 we looked at various sorting algorithms and analysed their time complexities.

- Bubble sort - $\Theta(n^2)$
- Quicksort - $\Theta(n \log n)$ on average but $\Theta(n^2)$ in the worst case
- Merge Sort, Heap Sort - $\Theta(n \log n)$ in the worst case

The question we want to answer is "is it possible to do any better?"

- What do we mean by “do better?”
- An algorithm gives an “upper bound” for a problem but tells us nothing about whether another faster algorithm might exist.
- We will look at how to prove a lower bound for sorting.
- Then we will show how to “beat” this lower bound.
Sorting revisited

- In COMS12100 we looked at various sorting algorithms and analysed their time complexities.
  - Bubble sort - $\Theta(n^2)$
  - Quicksort - $\Theta(n \log n)$ on average but $\Theta(n^2)$ in the worst case
  - Merge Sort, Heap Sort - $\Theta(n \log n)$ in the worst case
- The question we want to answer is "is it possible to do any better?"
  - What do we mean by “do better?”
  - An algorithm gives an “upper bound” for a problem but tells us nothing about whether another faster algorithm might exist.
  - We will look at how to prove a lower bound for sorting.
  - Then we will show how to “beat" this lower bound.
In order to prove a lower bound for a problem we need to somehow show that no algorithm can be faster than our bound in the worst case. This is different from seeing how fast a particular algorithm takes to run.

To prove a lower bound we need to show that for any program $P$ that correctly sorts all inputs, we can find an input such that $P$ takes time $\geq cn \log n$ for some $c$ (and for all $n$ larger than some constant).

We have to define carefully which operations we want to count. We have to define the computational model.

All the sorting algorithms we have seen so far have worked by comparing pairs of elements in the input. We will count only comparison operations and show that $\Omega(n \log n)$ comparisons are always needed in the worst case.

But is there any alternative to comparison sorting? We will see the answer to this question is yes.
Lower bounds for sorting

- In order to prove a lower bound for a problem we need to somehow show that no algorithm can be faster than our bound in the worst case. This is different from seeing how fast a particular algorithm takes to run.

- To prove a lower bound we need to show that for any program $P$ that correctly sorts all inputs, we can find an input such that $P$ takes time $\geq cn \log n$ for some $c$ (and for all $n$ larger than some constant).

- We have to define carefully which operations we want to count. We have to define the computational model.

- All the sorting algorithms we have seen so far have worked by comparing pairs of elements in the input. We will count only comparison operations and show that $\Omega(n \log n)$ comparisons are always needed in the worst case.

- But is there any alternative to comparison sorting? We will see the answer to this question is yes.
Lower bounds for sorting

- In order to prove a lower bound for a problem we need to somehow show that no algorithm can be faster than our bound in the worst case. This is different from seeing how fast a particular algorithm takes to run.

- To prove a lower bound we need to show that for any program $P$ that correctly sorts all inputs, we can find an input such that $P$ takes time $\geq cn \log n$ for some $c$ (and for all $n$ larger than some constant).

- We have to define carefully which operations we want to count. We have to define the computational model.

- All the sorting algorithms we have seen so far have worked by comparing pairs of elements in the input. We will count only comparison operations and show that $\Omega(n \log n)$ comparisons are always needed in the worst case.

- But is there any alternative to comparison sorting? We will see the answer to this question is yes.
Lower bounds for sorting

- In order to prove a lower bound for a problem we need to somehow show that no algorithm can be faster than our bound in the worst case. This is different from seeing how fast a particular algorithm takes to run.

- To prove a lower bound we need to show that for any program $P$ that correctly sorts all inputs, we can find an input such that $P$ takes time $\geq cn \log n$ for some $c$ (and for all $n$ larger than some constant).

- We have to define carefully which operations we want to count. We have to define the computational model.

- All the sorting algorithms we have seen so far have worked by comparing pairs of elements in the input. We will count only comparison operations and show that $\Omega(n \log n)$ comparisons are always needed in the worst case.

- But is there any alternative to comparison sorting? We will see the answer to this question is yes.
Lower bounds for sorting

- In order to prove a lower bound for a problem we need to somehow show that *no* algorithm can be faster than our bound in the worst case. This is different from seeing how fast a particular algorithm takes to run.

- To prove a lower bound we need to show that for any program $P$ that correctly sorts all inputs, we can find an input such that $P$ takes time $\geq cn\log n$ for some $c$ (and for all $n$ larger than some constant).

- We have to define carefully which operations we want to count. We have to define the computational *model*.

- All the sorting algorithms we have seen so far have worked by comparing pairs of elements in the input. We will count only comparison operations and show that $\Omega(n\log n)$ comparisons are always needed in the worst case.

- But is there any alternative to comparison sorting? We will see the answer to this question is yes.
Decision Trees

Sort \( < a_1, \ldots, a_n > \) (assume w.l.o.g. that all \( a_i \) are distinct)

Each internal node is labelled \( i : j \) for \( i, j \in \{1, 2, \ldots, n\} \)

- The left subtree shows subsequent comparisons if \( a_i \leq a_j \)
- The right subtree shows subsequent comparisons if \( a_i > a_j \)
Decision Trees

Sort < 7, 4, 6 >

Each internal node is labelled \( i : j \) for \( i, j \in \{1, 2, \ldots, n\} \)

- The left subtree shows subsequent comparisons if \( a_i \leq a_j \)
- The right subtree shows subsequent comparisons if \( a_i > a_j \)
Decision Trees

Sort $< 7, 4, 6 >$

Each internal node is labelled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$
Decision Trees

Sort < 7, 4, 6 >

Each internal node is labelled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i > a_j$
Decision Trees

Sort $< 7, 4, 6 >$

Each leaf contains a permutation to indicate the ordering of the input.
A decision tree can model the execution of any comparison sort.

- One tree for each input size $n$
- Tree contains all possible sequences of comparisons needed to sort the input
- The running time of the algorithm on a particular input is the length of the path taken
- Worst case is the height of the tree
Lower Bound for Decision Tree Model

**Theorem**

Any decision tree that can sort \( n \) elements must have height \( \Omega(n \log n) \).

**Proof.**

The tree must contain at least \( n! \) leaves as there are \( n! \) different permutations to choose from. A binary tree of height \( h \) has \( \leq 2^h \) leaves. Therefore \( n! \leq \text{number of leaves} \leq 2^h \).

\[
h \geq \log n! \geq \frac{n}{2} \log \left( \frac{n}{2} \right)
\]

\[
h \in \Omega(n \log n)
\]
Corollary

Merge sort and heap sort are asymptotically optimal comparison sorting algorithms.
Counting sort: No comparisons used at all

Input: $A[1, \ldots, n]$ where $A[j] \in \{1, 2, \ldots, k\}$

Output: $B[1, \ldots, n]$, sorted and a permutation of $A$

Auxiliary storage: $C[0, \ldots, k]$
Counting sort

\[\text{for } i \leftarrow 0 \text{ to } k \text{ do} \]
\[\quad C[i] \leftarrow 0;\]
\[\text{end}\]
\[\text{for } j \leftarrow 1 \text{ to } n \text{ do} \]
\[\quad C[A[j]] \leftarrow C[A[j]] + 1;\]
\[\text{end}\]
\[\triangleright C[i] \text{ now contains the number of elements equal to } i;\]
\[\text{for } i \leftarrow 2 \text{ to } k \text{ do} \]
\[\quad C[i] \leftarrow C[i] + C[i - 1];\]
\[\text{end}\]
\[\triangleright C[i] \text{ now contains the number of elements less than or equal to } i;\]
\[\text{for } j \leftarrow n \text{ downto } 1 \text{ do} \]
\[\quad B[C[A[j]]] \leftarrow A[j];\]
\[\quad C[A[j]] \leftarrow C[A[j]] - 1;\]
\[\text{end}\]
Counting sort

```plaintext
for i ← 0 to k do
    C[i] ← 0;
end
for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1;
end
▷ C[i] now contains the number of elements equal to i;
for i ← 2 to k do
    C[i] ← C[i] + C[i - 1];
end
▷ C[i] now contains the number of elements less than or equal to i;
for j ← n downto 1 do
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] - 1;
end
```
Counting sort

\[
\text{for } i \leftarrow 0 \text{ to } k \text{ do} \\
\quad C[i] \leftarrow 0; \\
\text{end}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad C[A[j]] \leftarrow C[A[j]] + 1; \\
\text{end}
\]

▷ \(C[i]\) now contains the number of elements equal to \(i\);

\[
\text{for } i \leftarrow 2 \text{ to } k \text{ do} \\
\quad C[i] \leftarrow C[i] + C[i - 1]; \\
\text{end}
\]

▷ \(C[i]\) now contains the number of elements less than or equal to \(i\);

\[
\text{for } j \leftarrow n \text{ downto } 1 \text{ do} \\
\quad B[C[A[j]]] \leftarrow A[j]; \\
\quad C[A[j]] \leftarrow C[A[j]] - 1; \\
\text{end}
\]
Counting sort

\[\text{for } i \leftarrow 0 \text{ to } k \text{ do}\]
\[\quad C[i] \leftarrow 0;\]
\[\text{end}\]
\[\text{for } j \leftarrow 1 \text{ to } n \text{ do}\]
\[\quad C[A[j]] \leftarrow C[A[j]] + 1;\]
\[\text{end}\]
\[\triangleright C[i] \text{ now contains the number of elements equal to } i;\]
\[\text{for } i \leftarrow 2 \text{ to } k \text{ do}\]
\[\quad C[i] \leftarrow C[i] + C[i - 1];\]
\[\text{end}\]
\[\triangleright C[i] \text{ now contains the number of elements less than or equal to } i;\]
\[\text{for } j \leftarrow n \text{ downto } 1 \text{ do}\]
\[\quad B[C[A[j]]] \leftarrow A[j];\]
\[\quad C[A[j]] \leftarrow C[A[j]] - 1;\]
\[\text{end}\]
Counting sort example

On blackboard...
Counting sort analysis

Sum the 4 different loops giving $\Theta(n + k)$ time in total.

\begin{verbatim}
for i ← 0 to k do
    C[i] ← 0;
end

The initialisation takes $\Theta(k)$ time;

for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1;
end

Building the array of element counts takes $\Theta(n)$ time;

for i ← 1 to k do
    C[i] ← C[i] + C[i - 1];
end

Accumulating the element count takes $\Theta(k)$ time;

for j ← n downto 1 do
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] - 1;
end

“Distribution" takes $\Theta(n)$ time;
\end{verbatim}
Counting sort analysis

Sum the 4 different loops giving $\Theta(n + k)$ time in total.

```plaintext
for i ← 0 to k do
    C[i] ← 0;
end
```
The initialisation takes $\Theta(k)$ time;

```plaintext
for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1;
end
```
Building the array of element counts takes $\Theta(n)$ time;

```plaintext
for i ← 1 to k do
    C[i] ← C[i] + C[i - 1];
end
```
Accumulating the element count takes $\Theta(k)$ time;

```plaintext
for j ← n downto 1 do
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] - 1;
end
```
“Distribution" takes $\Theta(n)$ time;
Counting sort analysis

Sum the 4 different loops giving $\Theta(n + k)$ time in total.

```plaintext
for i ← 0 to k do
    C[i] ← 0;
end
```
The initialisation takes $\Theta(k)$ time;

```plaintext
for j ← 1 to n do
    C[A[j]] ← C[A[j]] + 1;
end
```
Building the array of element counts takes $\Theta(n)$ time;

```plaintext
for i ← 1 to k do
    C[i] ← C[i] + C[i − 1];
end
```
Accumulating the element count takes $\Theta(k)$ time;

```plaintext
for j ← n downto 1 do
    B[C[A[j]]] ← A[j];
    C[A[j]] ← C[A[j]] − 1;
end
```
“Distribution” takes $\Theta(n)$ time;
Counting sort analysis

Sum the 4 different loops giving $\Theta(n + k)$ time in total.

\begin{verbatim}
for i <-> 0 to k do
    C[i] <-> 0;
end
The initialisation takes $\Theta(k)$ time;

for j <-> 1 to n do
    C[A[j]] <-> C[A[j]] + 1;
end
Building the array of element counts takes $\Theta(n)$ time;

for i <-> 1 to k do
    C[i] <-> C[i] + C[i - 1];
end
Accumulating the element count takes $\Theta(k)$ time;

for j <-> n downto 1 do
    B[C[A[j]]] <-> A[j];
    C[A[j]] <-> C[A[j]] - 1;
end
"Distribution" takes $\Theta(n)$ time;
\end{verbatim}
If $k \in \Theta(n)$ then the total running time of counting sort is $\Theta(n)$.

- But we proved an $\Omega(n \log n)$ lower bound! What happened?

**Answer:**

- We proved a lower bound for *comparison sorts*
- Counting sort is not a comparison sort
- In fact there is not a single comparison in it!
If $k \in \Theta(n)$ then the total running time of counting sort is $\Theta(n)$.

- But we proved an $\Omega(n \log n)$ lower bound! What happened?

**Answer:**

- We proved a lower bound for *comparison sorts*
- Counting sort is not a comparison sort
- In fact there is not a single comparison in it!
Stable sorting

A crucial property of counting sort is that it is stable.

- Why did we bother with the last two loops of counting sort?
- We want the sort to be stable.
- It preserves the order of equal elements

What other sorting algorithms are stable?

See blackboard...
Radix sort is possibly the oldest implemented sorting algorithm. It operates digit by digit from the least significant digit to the most significant one.

See blackboard...
Correctness of radix sort

We can prove the correctness of radix sort by induction on the digit position.

- Base case. Radix sort clearly is correct for single digit numbers.
- Inductive hypothesis. Assume the numbers are sorted by their $t - 1$ lowest order digits.
- Inductive step. Sort using digit $t$.
  - Two numbers that have the same digit $t$ preserve their original order by stability.
  - Two numbers that differ at digit $t$ are placed in the correct order.
Correctness of radix sort

We can prove the correctness of radix sort by induction on the digit position.

- **Base case.** Radix sort clearly is correct for single digit numbers.
- **Inductive hypothesis.** Assume the numbers are sorted by their $t - 1$ lowest order digits.
- **Inductive step.** Sort using digit $t$.
  - Two numbers that have the same digit $t$ preserve their original order by stability.
  - Two numbers that differ at digit $t$ are placed in the correct order.
Correctness of radix sort

We can prove the correctness of radix sort by induction on the digit position.

- Base case. Radix sort clearly is correct for single digit numbers
- Inductive hypothesis. Assume the numbers are sorted by their \( t - 1 \) lowest order digits
- Inductive step. Sort using digit \( t \)
  - Two numbers that have the same digit \( t \) preserve their original order by stability
  - Two numbers that differ at digit \( t \) are placed in the correct order
Correctness of radix sort

We can prove the correctness of radix sort by induction on the digit position.

- Base case. Radix sort clearly is correct for single digit numbers
- Inductive hypothesis. Assume the numbers are sorted by their $t-1$ lowest order digits
- Inductive step. Sort using digit $t$
  - Two numbers that have the same digit $t$ preserve their original order by stability
  - Two numbers that differ at digit $t$ are placed in the correct order
Correctness of radix sort

We can prove the correctness of radix sort by induction on the digit position.

- **Base case.** Radix sort clearly is correct for single digit numbers.
- **Inductive hypothesis.** Assume the numbers are sorted by their $t-1$ lowest order digits.
- **Inductive step.** Sort using digit $t$
  - Two numbers that have the same digit $t$ preserve their original order by stability.
  - Two numbers that differ at digit $t$ are placed in the correct order.
Running time of radix sort

- Assume counting sort is used as the auxiliary sorting algorithm
- Sort \( n \) numbers of \( b \) bits each
- A "digit" is \( r \leq b \) bits long
- If the numbers are 32 bits long and \( r = 8 \) then we can think of the input as having 4 digits in base \( 2^r \)
- What should we set \( r \) to?

See blackboard...
Running time of radix sort

Remember that counting sort takes $\Theta(n + k)$ time.

- Each call to counting sort takes $\Theta(n + 2^r)$ time
- $b/r$ calls are made to counting sort
- The total time is therefore

$$T(n, b) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

- How can we set $r$ to minimise this?
- Increasing $r$ means fewer passes but then the time for counting sort grows exponentially
Running time of radix sort

To help us get a feel for the problem we plot $r$ against the running time $(b/r)(n + 2^r)$ (setting all constant factors to 1).

![Graph showing the running time of radix sort for b=32, n=1000 and r=1...15]
Running time of radix sort

We want to minimise

\[ T(n, b) \in \Theta\left(\frac{b}{r}(n + 2^r)\right) \]

- Formally we should differentiate \((b/r)(n + 2^r)\) and set the result to 0 to minimise the function.
- However, we can get a good guess (which turns out to be correct) by remembering that \(n + 2^r \in \Theta(\max(n, 2^r))\). We set \(r = \log(n)\) so that \(2^r = n\).
- Therefore

\[ T(n, b) \in \Theta(n b / \log n) \]

- For numbers in the range from 0 to \(n^{d-1}\), we have \(b = d \log n \Rightarrow \) radix sort runs in \(\Theta(dn)\) time.
Summary

- Counting sort runs in $\Theta(n)$ time when the values to be sorted are less than $n$.
- Radix is fast for 32-bit numbers, for example, where only 4 passes of count sort and an auxiliary array of size $2^8 = 256$ are needed if we set $r = 8$.
- Merge sort or quicksort will need at least 11 linear time passes if there are, say, $\geq 2000$ values to be sorted.
- However, radix sort has poor memory locality and so a well tuned quicksort may be faster in practice.