Sorting in Linear Time?

Lower Bound for Comparison-Based Sorting

Lower bound for all sorting algorithms requires a precise definition of a **sorting algorithm**.

Comparison-based: elements can be compared with other elements, but not participate in other operations.

- **Basic action**: compare two elements in input and make a choice between two ways to proceed.
- **Answer**: the arrangement which must be made to obtain sorted order.
- **ID for elements**: their original position (index) in input.

If we start by annotating all input elements with their original position, we can always keep track of which two IDs are compared in a concrete algorithm.

Annotation of input:

$F, A, C, B, E, D \rightarrow (F, 1), (A, 2), (C, 3), (B, 4), (E, 5), (D, 6)$

Decision Trees

Precise model that defines the concept of "comparison-based sorting algorithms":

- Labels for inner nodes: IDs (i.e., original index in input) for two input elements that are compared.
- Labels for leaves (answer when the algorithm stops): which arrangement should be made to obtain sorted order (specified with list of IDs, i.e., of original indexes for input elements).

Worst-case runtime: longest root-leaf path = tree height.

Note: Insertionsort, selectionsort, mergesort, quicksort, heapsort can all be described this way.

Lower Bound for Comparison-Based Sorting

For a fixed collection of n elements, there are $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \ldots \cdot n$ different inputs (order of elements).

If the algorithm (tree) must be able to sort all these, there must be at least $n!$ leaves - otherwise there will be two different inputs that lead to the same answer, and for one input the answer must be wrong.

A tree of height h has at most 2^h leaves (since the full tree of height h has that).

$2^h \geq \text{number of leaves} \geq n!$

$h \geq \log(n!) = \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n) = \log(1) + \log(2) + \ldots \log(n/2) \cdot \ldots + \log(n) \geq \frac{n}{2} \cdot \log(\frac{n}{2}) = \frac{n}{2}(\log(n) - 1)$

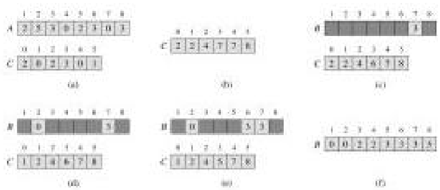
So worst-case runtime = tree height $h = \Omega(n \log n)$

Counting Sort

Assumes that keys are integers, of size up to k. This allows elements to be used as array indices (≠ using comparisons on elements).

> **Counting sort**: Sorts n integers of size between 0 and k (inclusive).
>
> - Input array A (length n)
> - Output array B (length n)
> - Array of counters for each possible element value: C (length $k+1$)

The following image shows an example of counting sort and how it works:



Time: $O(n+k)$

Note: stable, i.e., elements with equal values retain their mutual space (since the last loop runs backwards through A (and B for each value)).

Radix Sort

> **Radix sort**: Sorts n integers all with d digits in base (radix) k. (i.e. the digits are integers in $0, 1, 2, \ldots, k-1$)

In the figure, there are 7 integers with 3 digits in base 10.

`Radix-Sort(A,d)`

```
for i = 1 to d
    use a stable sort to sort A on digit i from right
```

Time: $O(d(n+k))$ if Counting Sort is used in the for-loop.

Correctness: After the i'th iteration of the for-loop, A is sorted if one only looks at the i digits most to the right.

Radix Sort Example

Example: integers in the decimal system with width 12: 486 239 123 989

Countingsort sorts these in time $O(n+10^{12})$. This is $O(n)$ if $n \geq 10^{12} = 1,000,000,000,000$

See as 2-digit numbers in base 10^6 (note: sorted order is the same): 486 239 123 989

Radixsort sorts these in time $O(2(n + 10^6))$. This is $O(n)$ if $n \geq 10^6 = 1,000,000$

See as 4-digit numbers in base 10^3 (note: sorted order is the same): 486 239 123 989

Radixsort sorts these in time $O(4(n + 10^3))$. This is $O(n)$ if $n \geq 10^3 = 1,000$

Radix Sort Example 2

Example: integers in the binary system with width 32: 11011001 10011000 01101000 10110101

Countingsort sorts these in time $O(n + 2^{32})$. This is $O(n)$ if $n \geq 2^{32} = 4,294,967,296$

See as 2-digit numbers in base 2^{16} (note: sorted order is the same): 11011001 10011000 01101000 10110101

See as 4-digit numbers in base 2^8 (note: sorted order is the same): 11011001 10011000 01101000 10110101

Radixsort sorts these in time $O(4(n + 2^8))$. This is $O(n)$ if $n \geq 2^8 = 256$