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 to other elements but cannot participate in other operations.

- **Basic operation:** Comparison of two elements in the input.
- **Basic outcome:** The arrangement needed to achieve sorted order.
- **ID of elements:** Their original position (index) in the input.

Note: if we start by annotating all input elements with their original position, in a specific algorithm, we can always track which two IDs are being compared in a specific algorithm.

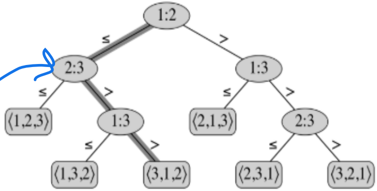
Annotation of input:

51, 27, 99, 61, 18, 37, . . . → (51,1) , (27,2) , (99,3) , (61,4) ,(18,5) , (37,6) , . . .

Decision Trees

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

numbers here are the original indicies



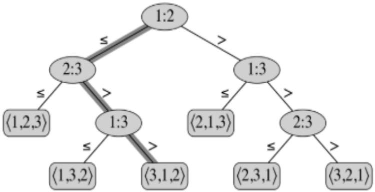
Labels for internal nodes: IDs (i.e. original index in input) of two input elements being compared.

Labels for Leaves (answer when the algorithm stops):The arrangement needed to achieve sorted order (given as a list of IDs, i.e., the original indices of input elements).

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

Note: Insertion sort, selection sort, merge sort, quick sort, heap sort can all be described like this.

Lower bound for comparison-based sorting



For a fixed set of n elements, there are n! = 1. 2. 3. 4. 5. …. .n different inputs (permutations of elements).

If the algorithm (the tree) must be able to sort all of these, there must be at least n! leaves—otherwise, there would be two different inputs leading to the same output, and for one of those inputs, the answer would have to be incorrect.

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

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

$$h \geq log(n!) = log\ (1\ .\ 2\ .\ 3\ \ldots\ .\ n)$$

$$log(1) + log(2) + \ldots + log(\tfrac{n}{2}) + \ldots + log(n) \geq \tfrac{n}{2}\ .\ log(\tfrac{n}{2}) = \tfrac{n}{2}\,(log(n) - 1)$$

So worst-case running time = height of the tree $h = \Omega(n\ log.n)$

Counting sort

a know max value

Assume that the keys are integers, with values up to **k**. This allows elements to be used as array indices (≠ using comparisons on elements).

Counting sort: Sorts **n** integers with values between **0** and **k** (inclusive).

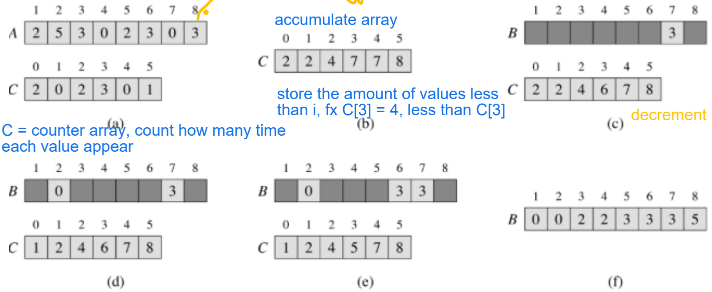
Input array: A (length **n**)

Output array: B (length **n**)

start form the end,3 see that C[3] = 7 go to position 7, on our extra array

Counter array for each possible element value: **C** (length **k + 1**)

count



accumulate array

store the amount of values less than i, fx C[3] = 4, less than C[3]

decrement C[3] = 6

C = counter array, count how many time each value appear

Counting sort

```
COUNTING-SORT(A, n, k)
      let B[1 : n] and C[0 : k] be new arrays
O(k)  for i = 0 to k
          C[i] = 0
O(n)  for j = 1 to n
          C[A[j]] = C[A[j]] + 1
      // C[i] now contains the number of elements equal to i.
O(k)  for i = 1 to k
          C[i] = C[i] + C[i − 1]
      // C[i] now contains the number of elements less than or equal to i.
      // Copy A to B, starting from the end of A.
O(n)  for j = n downto 1
          B[C[A[j]]] = A[j]
          C[A[j]] = C[A[j]] − 1    // to handle duplicate values
      return B
```

= O(2k+2n) = O(2(k+n)) constants doesn't matter in time complexity, O(n+k)

Time: O(n+k)

efficient when the range of value, is smaller than the total amount of elements

Note: Stable, meaning elements with the same values retain their relative order (since the last loop runs backward through A (and B for each value)).

Radix sort

Radix sort: Sorting n integers, each with d digits in base (radix) k. (i.e. the digits are integers in {0,1,2, . . . ,k −1})

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

RADIX-SORT(A,d)

O(d) **for** $i = 1$ **to** d

O(n+k) use a stable sort to sort A on digit i from right like count sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Time: O(d(n+k)) if Counting Sort is used in the **for**-loop d = base, n = numbers, k = base (here 10)

simplyfices to O(n), d and k are considered constants

Correctness:

After the **i**-th iteration of the for-loop, **A** is sorted if you only look at the **i** rightmost digits.

Radix sort

Example: Integers in the decimal system with a width of 12

486 239 123 989

Counting Sort sorts these in time $O(n+10^{12})$

This is $O(n)$ if $n \geq 10^{12}$=1.000.000.000.000

View 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

View 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: Integers in the binary system with a width of 32 (i.e., binary numbers with 32 bits)

11011001 10011000 01101000 10110101

Counting sort sorts these in time $O(n + 2^{32})$. This is $O(n)$ if $n \geq 2^{32} = 4.294.967.296$

View as 2-digit numbers in base 2^{16} (note: sorted order is the same)

11011001 10011000	01101000 10110101

Radixsort sorts these in time $O(2(n+2^{16}))$. This is $O(n)$ if $n \geq 2^{16} = 65.536$

View 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 $n \geq 2^{8} = 256$