6.006 Lecture 10: Sorting III

CLRS

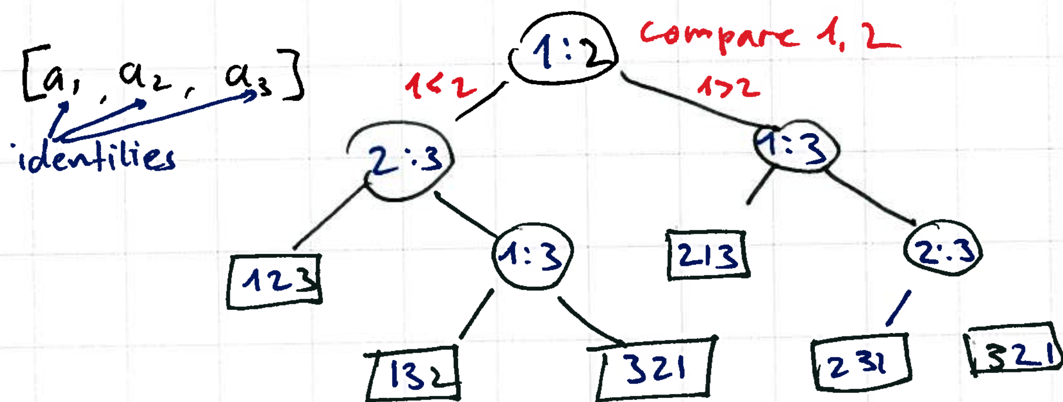
- Decision-tree lower bound for sorting — 8.1
- Sorting in linear time: counting sort — 8.2
- Radix Sort if we have time left — 8.3

Can we sort in $o(n \lg n)$ (faster than $\Theta(n \lg n)$)?

Algorithms so far were based on comparing elements; no other data dependent decisions (except for dependency on n)

Think of the elements as having an identity $1, 2, 3, \ldots n$

In each step, the algorithm compares two elts, changes its state (variables, locations of elts in memory) at the end, the state indicates the sorted order.



Theorem: Any decision tree that can sort n elements must have height $\Omega(n \lg n)$

Proof: Tree must have $\geq n!$ leaves (each a different permutation of $1, 2, \cdots n$

Tree is binary $\Rightarrow$ at most 2^h leaves for height h

$$n! \leq 2^h$$
$$\lg(n!) \leq \lg(2^h) = h$$

$$h \geq \lg(n!) \geq \lg\left(\left(\frac{n}{e}\right)^n\right) \geq n \lg n - n \lg e = \Omega(n \lg n)$$

stirling

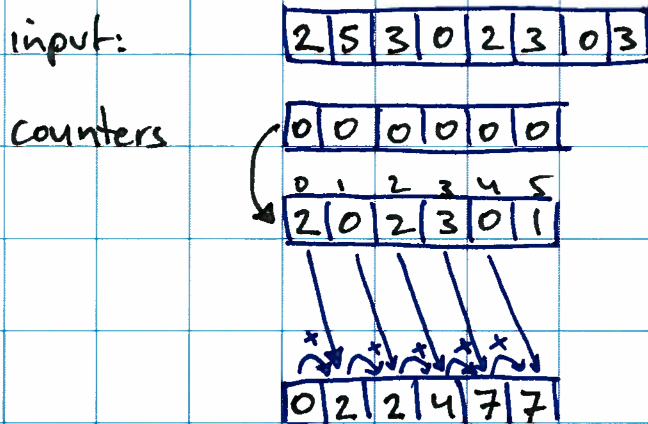
So, can we sort in $o(n \lg n)$?

No if we rely on comparisons & do not assume anything about the input.

Yes if we sort integers that are not too huge (that is, integers that are $O(n)$ or $O(n^3)$ etc.), or if we assume something on input distribution (e.g. uniform distribution in $[0, 1]$).

Counting Sort

- Sorts integers ≥ 0
- Runs in $O(n)$ work & space *if* inupt numbers are $O(n)$
- Key idea: use input keys to index into an array.

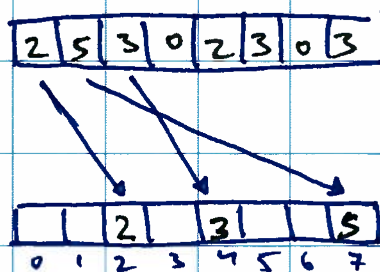


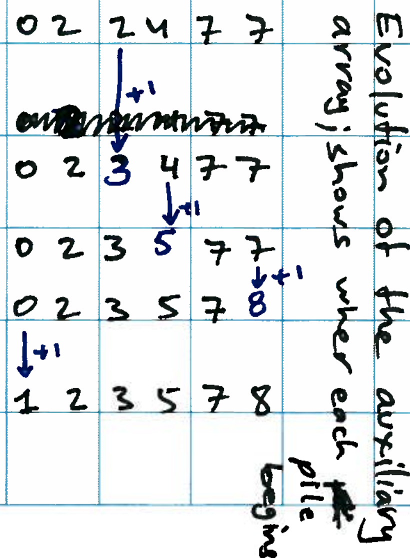
How many of each value?
Scan input once (loop) to find out.

compute prefix sums
(if we represent first $\emptyset$ implicitly, can do this in place; textbook)

Now we know where each pile of same value starts:
the zeros start at $\emptyset$, the ones at 2, the fives at 7, ...

Lets put them in place





Running time of counting sort

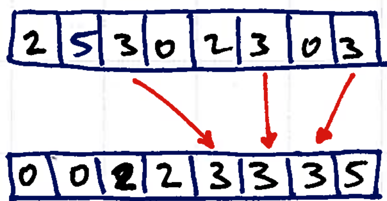
- Let the inputs be integers between 0 and k
- Initializing the counters array: $\Theta(k)$
- Scanning the input to count occurances of values: $\Theta(n)$
- Computing prefix sums: $\Theta(k)$
- Arranging inputs into sorted output: $O(n)$

Total: $O(n+k)$

Efficient when k is not much larger than n.

Counting sort is stable

- The output ordering of two same-value keys is the same as their input ordering



- This allows us to use counting sort as a subroutine in Radix Sort, which sorts integers in larger ranges in linear time.

Radix Sort

- Key idea: view input integers as multidigit numbers in some convenient base (10 for exposition, larger in practice, e.g. base 1024, or base 1,000,000, etc).
- Use counting sort on each digit
- Intuitively, we might sort on Most significant digit first; this is not how Radix sort works; too many invocation of counting sort.

ex: (sort) (sort separately) (sort, but we are already done here.)

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

- Radix sort does it the other way around

			sorted
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

- Stability of counting sort is key; would not work otherwise.