>>

Sorting (ii)

∧ >>

❖ Summary of Sorting Methods

Sorting = arrange a collection of *N* items in ascending order ...

Elementary sorting algorithms: $O(N^2)$ comparisons

- selection sort, insertion sort, bubble sort

Advanced sorting algorithms: $O(N \log N)$ comparisons

- quicksort, merge sort, heap sort (priority queue)

Most are intended for use in-memory (random access data structure).

Merge sort adapts well for use as disk-based sort.

❖ ... Summary of Sorting Methods

Other properties of sort algorithms: stable, adaptive

Selection sort:

- stability depends on implementation
- not adaptive

Bubble sort:

- is stable if items don't move past same-key items
- adaptive if it terminates when no swaps

Insertion sort:

- stability depends on implementation of insertion
- adaptive if it stops scan when position is found

<< ^ >>

❖ ... Summary of Sorting Methods

Other properties of sort algorithms: stable, adaptive

Quicksort:

- easy to make stable on lists; difficult on arrays
- can be adaptive depending on implementation

Merge sort:

- is stable if merge operation is stable
- can be made adaptive (but version in slides is not)

Heap sort:

- is not stable because of top-to-bottom nature of heap ordering
- adaptive variants of heap sort exist (faster if data almost sorted)

<< ^ >>

❖ Lower Bound for Comparison-Based Sorting

All of the above sorting algorithms for arrays of *n* elements

- have comparing whole keys as a critical operation

Such algorithms cannot work with less than *O(n log n)* comparisons

Informal proof (for arrays with no duplicates):

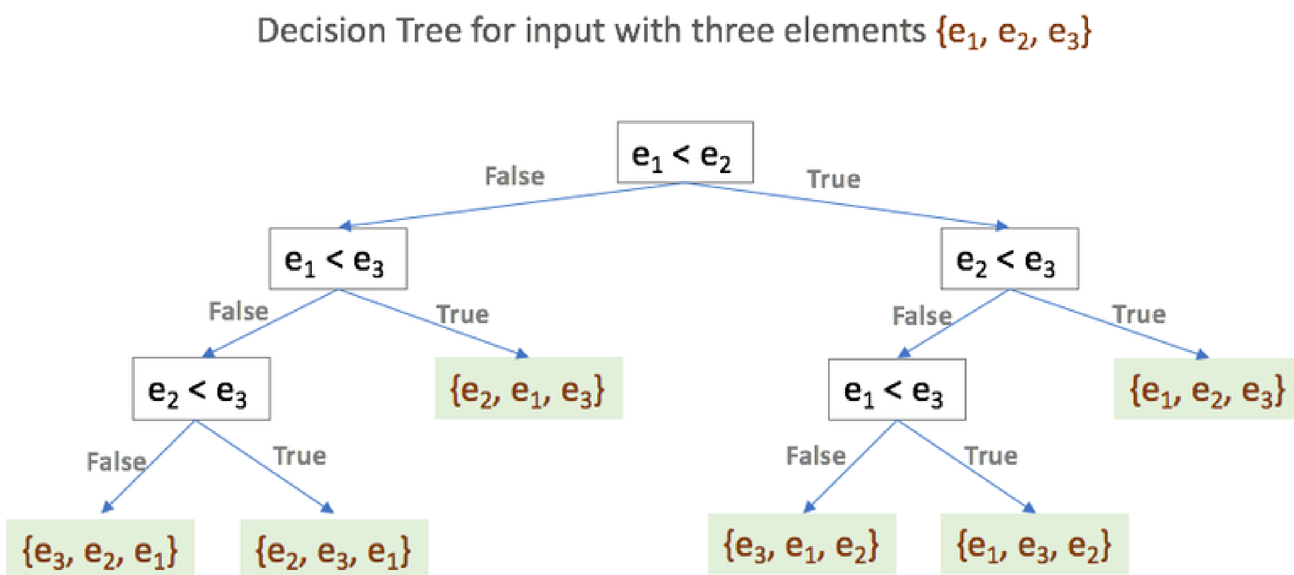
- there are *n!* possible permutation sequences
- one of these possible sequences is a sorted sequence
- each comparision reduces # possible sequences to be considered

(continued ...)

<< ∧ >>

❖ ... Lower Bound for Comparison-Based Sorting

Can view sorting as navigating a decision tree ...



(continued ...)

<< ∧ >>

❖ ... Lower Bound for Comparison-Based Sorting

Can view the sorting process as

- following a path from the root to a leaf in the decision tree
- requiring one comparison at each level

For n elements, there are $n!$ leaves

- height of such a tree is at least $log_2(n!)$
  ⇒ number of comparisions required is at least $log_2(n!)$

So, for comparison-based sorting, lower bound is $\Omega(n\ log_2 n)$.

Are there faster algorithms not based on whole key comparison?

<< ^ >>

❖ Radix Sort

Radix sort is a non-comparative sorting algorithm.

Requires us to consider a key as a tuple $(k_1, k_2, ..., k_m)$, e.g.

- represent key 372 as (3, 7, 2)
- represent key **`"sydney"`** as (s, y, d, n, e, y)

Assume only small number of possible values for k_i, e.g.

- numeric: 0-9 ... alpha: a-z

If keys have different lengths, pad with suitable character, e.g.

- numeric: 123, 002, 015 ... alpha: **`"abc"`**, **`"zz␣"`**, **`"t␣␣"`**

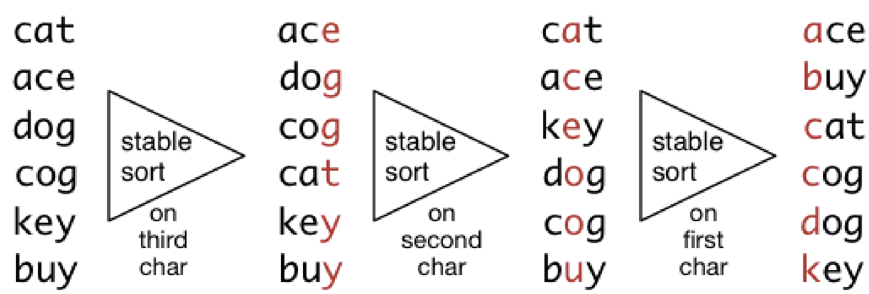
<< ∧ >>

❖ ... Radix Sort

Radix sort algorithm:

- stable sort on k_m,
- then stable sort on $k_{(m-1)}$,
- continue until we reach k_1

Example:



<< ∧ >>

❖ ... Radix Sort

Stable sorting (bucket sort):

```
// sort array A[n] of keys
// each key is m symbols from an "alphabet"
// array of buckets, one for each symbol
for each i in m .. 1 do
   empty all buckets
   for each key in A do
      append key to bucket[key[i]]
   end for
   clear A
   for each bucket in order do
      for each key in bucket do
         append to array
      end for
   end for
end for
```

❖ ... Radix Sort

Example:

- `m` = 3, alphabet = {'a', 'b', 'c'}, B[ ] = buckets
- A[ ] = {"abc", "cab", "baa", "a␣␣", "ca␣"}

After first pass (`i` = 3):

- B['a'] = {"baa"}, B['b'] = {"cab"}, B['c'] = {"abc"}, B['␣'] = {"a␣␣","ca␣"}
- A[ ] = {"baa", "cab", "abc", "a␣␣", "ca␣"}

After second pass (`i` = 2):

- B['a'] = {"baa","cab","ca␣"}, B['b'] = {"abc"}, B['c'] = {}, B['␣'] = {"a␣␣"}
- A[ ] = {"baa", "cab", "ca␣", "abc", "a␣␣"}

After third pass (`i` = 1):

- B['a'] = {"abc","a␣␣"}, B['b'] = {"baa"}, B['c'] = {"cab","ca␣"}, B['␣'] = {}
- A[ ] = {"abc", "a␣␣", baa", "cab", "ca␣"}

<< ^

❖ ... Radix Sort

Complexity analysis:

- array contains n keys, each key contains m symbols
- stable sort (bucket sort) runs in time $O(n)$
- radix sort uses stable sort m times

So, time complexity for radix sort = $O(mn)$

Radix sort performs better than comparison-based sorting algorithms

- when keys are short (small m) and arrays are large (large n)

Produced: 23 Jul 2020