Comparison sort

comparison operation : ← properties of a total order

1. transitivity : if $a \leq b$ and $b \leq c$ then $a \leq c$
2. total relation : for all a and b, either $a \leq b$ or $b \leq a$

A **comparison sort** is a type of sorting algorithm

Input:

- elements
- a comparison operation

  determines which of two elements should occur first in the final sorted list
  (often a "less than or equal to" operator)

Output

- list of elements
  order determined by comparison operation

Possible: $a \leq b$ and $b \leq a$; in this case either may come first in the sorted list.

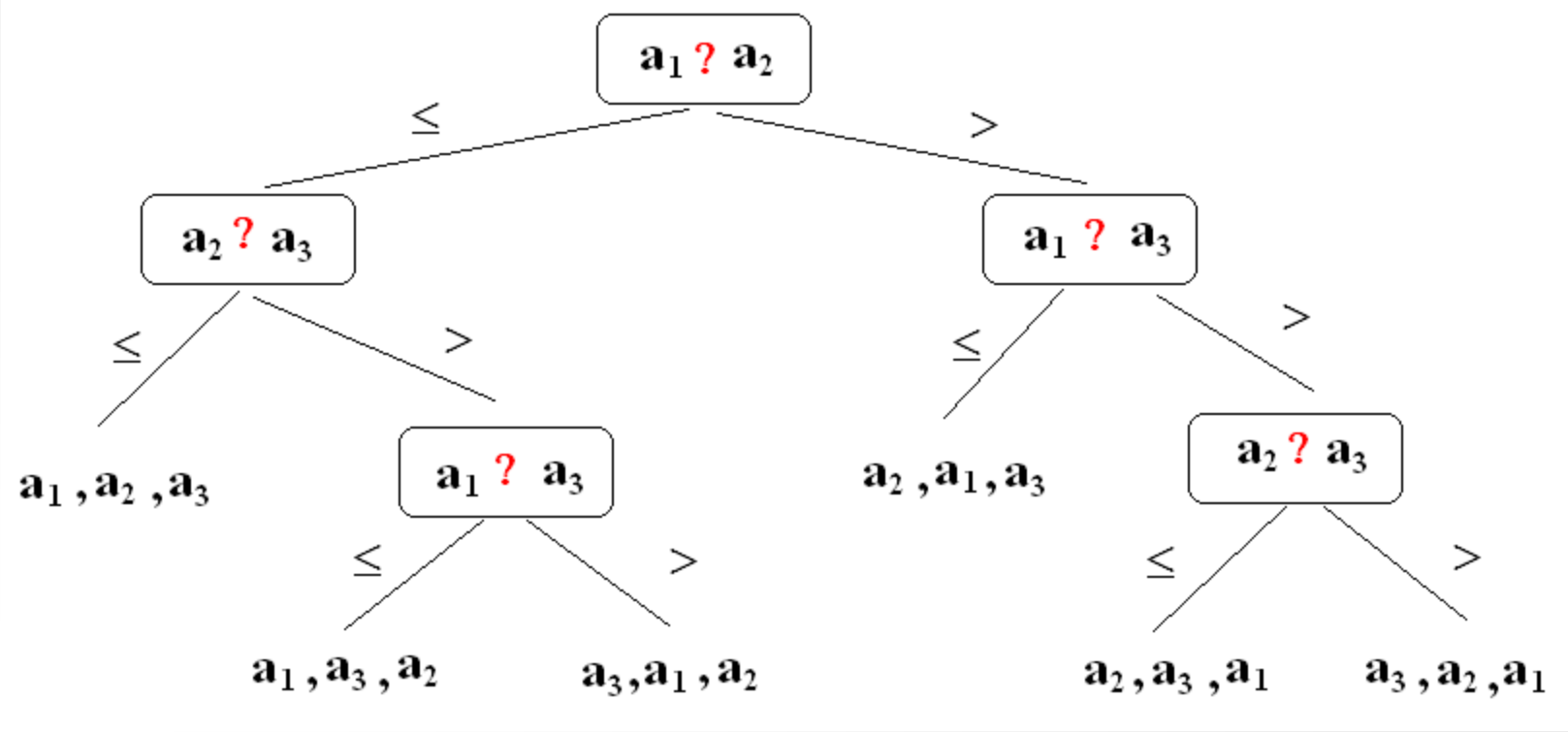
Stable sort: the input order determines the sorted order in case of $a \leq b$ and $b \leq a$

Decision tree

A decision tree represents the comparisons performed by a sorting algorithm when it operates on an input of a given size

- Example:

  A decision tree for insertion sort operating on 3 elements



Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\theta\left(\frac{1}{n}\right)\right)$$

(Cormen)

=> $\mathbf{\log(n!) = \Omega(n \log n)}$

Lower bounds for comparison sort

- **Theorem**

  Any decision tree that sorts n elements

  has height (>=) $\Omega(n \lg n)$.
- **Consequence**

  Heapsort is asymptotically optimal comparison sort

  (and mergesort)

Counting sort

- assumes that each of the n input elements is an integer in the range 1 to k,
- the overall time is $O(k + n)$.

  when $k = O(n)$, the sort runs in $O(n)$ time
- uses a temporary array

Arrays used in subalg.:

- A[1 ... n] original unsorted array
- B[1 ... n] array to hold sorted output
- C[1 ... k] working array to hold counts

Subalg. CountingSort (A, B, k)

```
for i := 1 to k  do        C[i] := 0          endfor
for j := 1 to length(A) do C[A[j]] := C[A[j]] + 1 endfor
// C[i] now contains the nr. of elem. equal to i

for i := 2 to k do C[i] := C[i] + C[i-1] endfor
// C[i] now contains the nr. of elem. less than or equal to i

for j := length(A) downto 1 do
    B[C[A[j]]] := A[j]
    C[A[j]] := C[A[j]] - 1
endfor
```

EndCountingSort

Ex: 7,1,3,1,2,4,5,7,2,4,3

Radix Sort

- sorts integers by processing individual digits.
- apply to string of comparable elements
  - integers represented as strings of digits
  - strings
  - date: (year, month, day)

Steps of radix sort algorithm

- sort by *least significant* digit first

  into groups

  but (otherwise) keep the original order
- combine them
- repeat the grouping process with each more significant digit

Radix sort

Subalg. radixSort(*A*, *d*) (Steps !!)

for i := *least_signif_digit* to *most_signif_digit* do

@ do use a stable sort to sort array *A* on digit *i*

endfor

endRadixSort

- stable sort

maintain the relative order of records with equal keys

Ex: 34, 12, 42, 32, 44, 41, 34, 11, 32, 23

Bucket Sort

- works by partitioning an array into a number of buckets

Steps of BucketSort method:

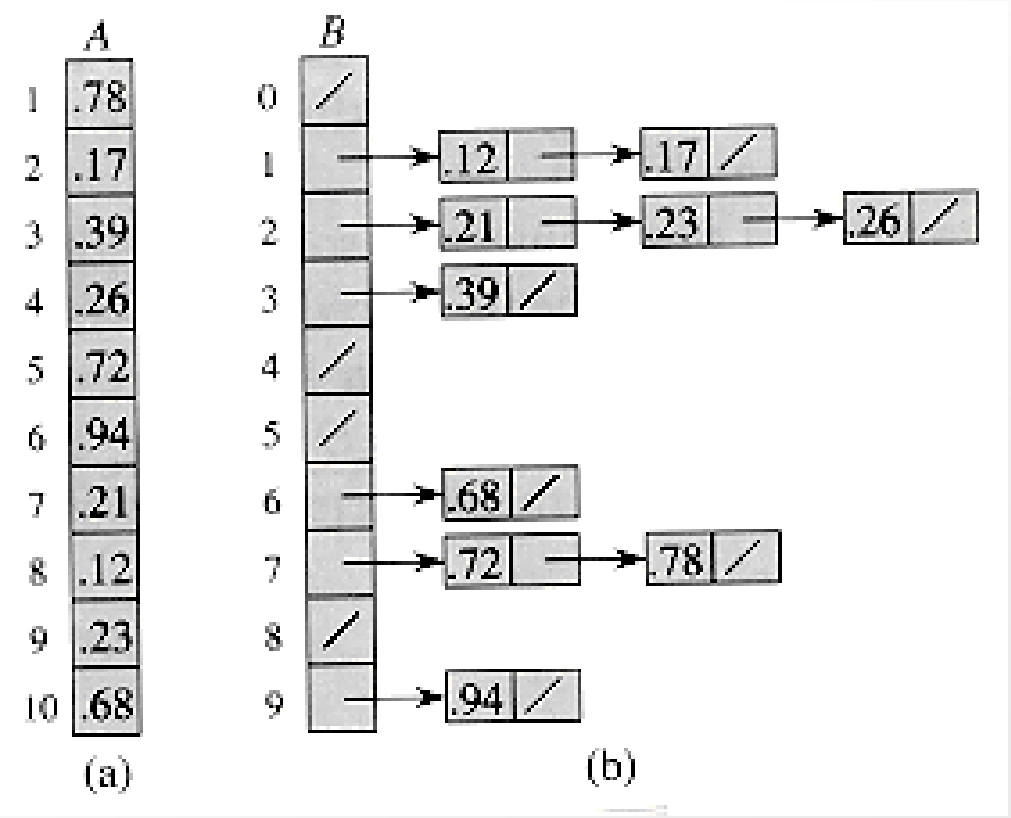
1. Set up an array of initially empty *buckets*.
2. Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Visit the buckets in order and put all elements back into the original array

Bucket Sort

Given:

- an *n*-element array *A*
- that each element *A*[*i*] in the array satisfies

$0 <= A[i] < 1.$



Bucket Sort

```
Subalg. bucketSort(A)
n := length(A)
for i := 1 to n do
    @ insert A[i] into bucket list B[floor(n*A[i])]
endfor
for i := 0 to n –1 do
    @ sort list B[i]
endfor
@concatenate the lists B[0], B[1], . . . , B[n - 1] // in order
endBucketSort.
```

```
unsigned int const m =  … //
…
void BucketSort(unsigned int *a, unsigned int n)
{
    int buckets[m];
    for( unsigned int j=0; j<m; ++j)
        buckets[j]=0;
    for (unsigned int i=0;  i<n; ++i)
        ++buckets[a[i]];
    for (unsigned int i=0, j=0; j<m; ++j)
        for (unsigned int k =buckets[j]; k>0; --k)
            a[i++] = j;
}
```

Ex: 7,1,3,1,2,4,5,7,2,4,3