Comparison sort

comparison operation : 🡐 properties of a total order

1. transitivity : if a ≤ b and b ≤ c then a ≤ c
2. total relation : for all a and b, either a ≤ b or b ≤ 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 ≤ b and b ≤ a; in this case either may come first in the sorted list.

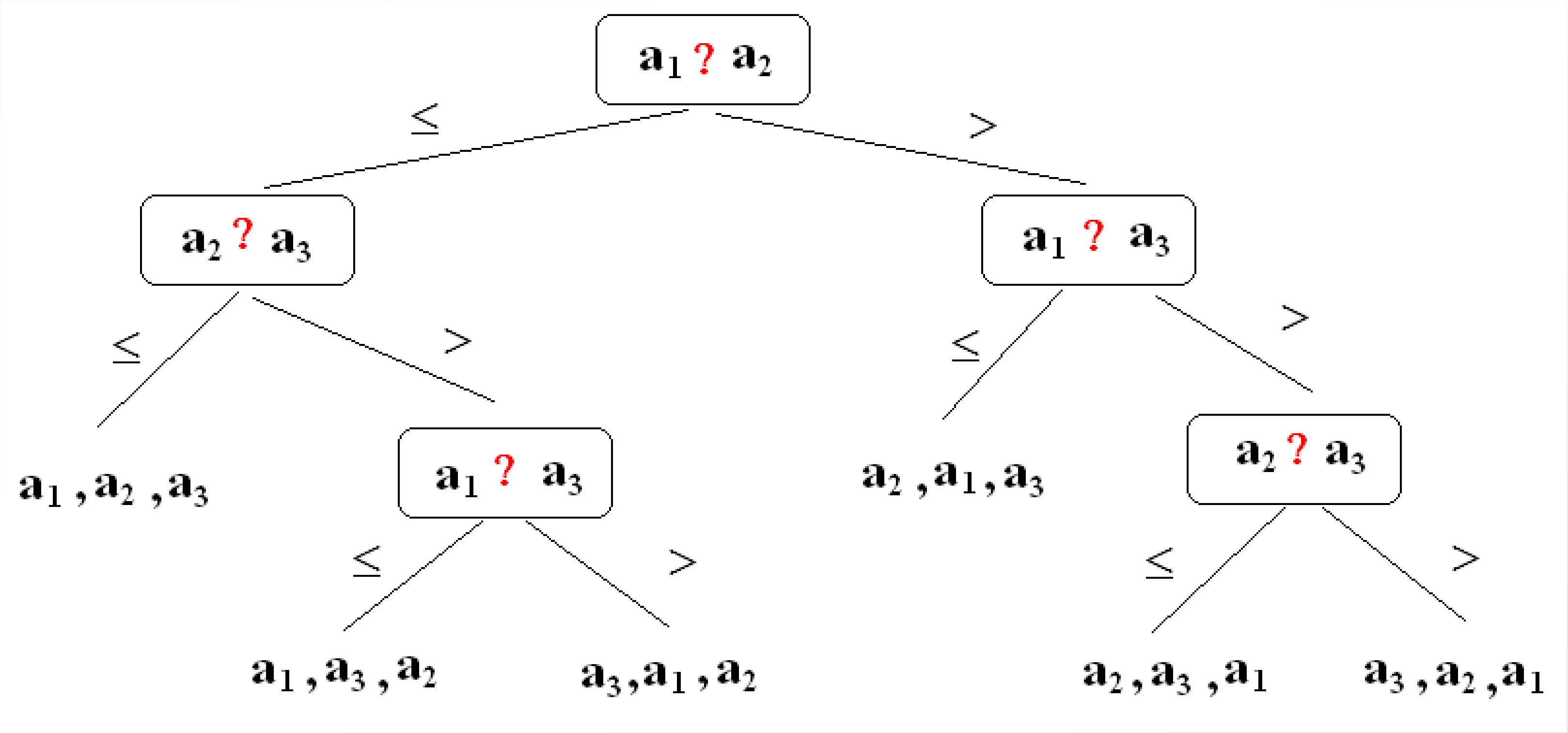
Stable sort: the input order determines the sorted order in case of a ≤b and b ≤a

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

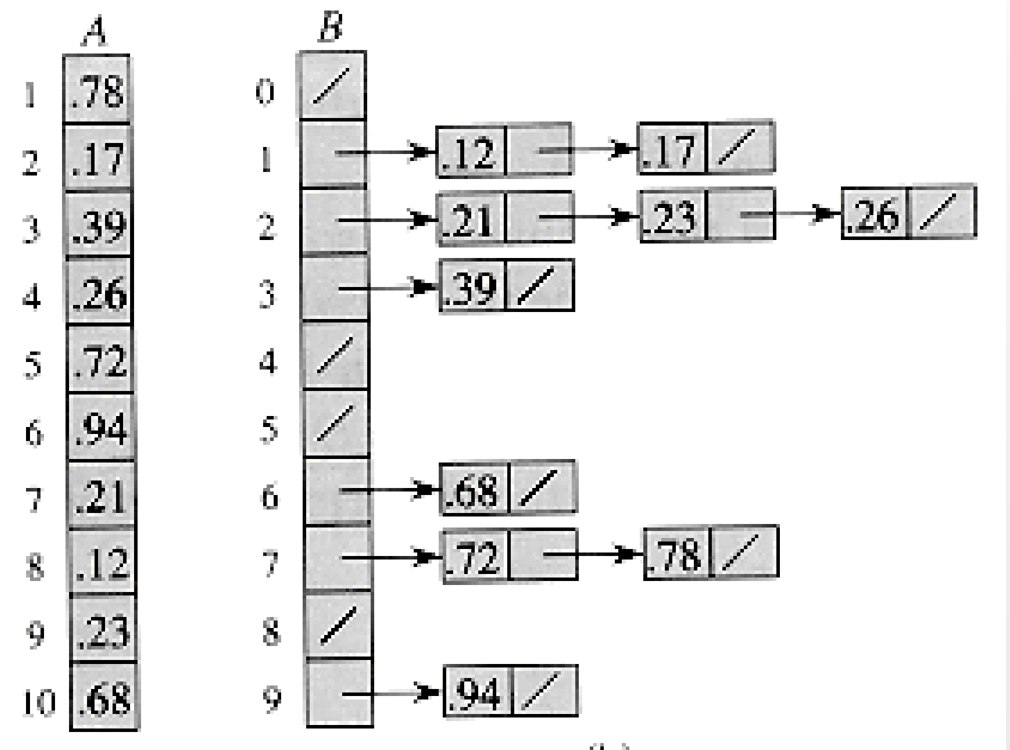
Bucket Sort

Given:

- an n-element array A

- that each element $A[i]$ in the array satisfies

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



(a) (b)

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

Cheat sheet - (Official!!)

- both sides of one sheet of paper
- A4 paper size
- your name and group on it
- you can write anything as long as is **written by your own hand**
- at the end of the exam, the **cheat sheet** will be also delivered

Please respect conditions stated here;
otherwise, you will not be allowed with the cheat sheet.

No other additional resources!

What kind of subjects ?

Please see any question/problem/exercise discussed/presented on slides, lecture or seminar classes.

The final exam subjects will not consist of complex problems

- no problems as project problems

Complexity analisys

Ex: Consider Dynamic Vector:

a) D.S.
b) specify and design addLast, reserve
c) inserting a series of elements into a vector is a ***linear or quadratic*** time operation? Justify!

ADTs

- **(Short) ADTs**
  Stack, Queue, Forward Iterator, …
- **(Short) ADTs with some restrictions**
  ADT Forward Iterator.
  - use operations: hasNext, next
    next – move to the next element and return it
- **Part of some ADTs**
  ex.: modifiers operations

DS

- **representation (for a given ADT)**

Operations / subalg.

- **Specification & pseudocode**
- **Design (& with some restrictions)**
  Consider insertBefore operation for a singly linked list.
  Can the operation be done with list traversal, without or both?
  (Specify consequences & pseudocode for possible operations).

- **Give subalg. for similar with the studied subalg.**
  adapt studied algorithms
  some algorithms were only only discussed
  specify and design heapSort subalg. Present used d.s.

- **Use the subalg.**

Give (non-trivial) examples
Illustrate how subalg. works

Give 3 different degenerated BST containing values 1, 2, 3, 4
Insert a given values into a given RB-tree
describe the insert cases to be applied; show the result

True/false questions

Example:

Subject nr. 1

1. ADT Queue

2. Define a forward iterator for a binary tree; get elements on levels, left to right. (Do not use recursive subalgorithms)

a) present traversal idea (ex.: original subalg.+ how to split code)

b) representation & pseudocode;
 specify any used or implemented operations

c) Print all the elements stored in a binary tree; use the iterator.

- specification & pseudocode
- illustrate/describe what happens if the tree is empty

3. Open addressing and double hashing.

Suppose we have a hash table of size 13, and:

$h_1(k) = k \text{ mod } 13$

$h_2(k) = 1 + (k \text{ mod } 11)$.

Illustrate the insertion of values 10, 22, 31, 4, 15, 17, 18, 19 into an initially empty hash table. (Present intermediary computations.)

4. An (almost) complete tree with height h has:

a) 2^h nodes b) 2^{h+1} nodes

c) between 2^h and 2^{h+1} nodes

d) between 2^h and 2^{h+1}-1 nodes