Data Structures & Algorithms

Lecture 4: Linear Sorting

Chapter 8

Many slides by Kevin Buchin

Tips

- Analysis of recursive algorithms:
  find the recursion and solve

- Analysis of loops: summations

- Some standard recurrences and sums:
  - $T(n) = 2T(n/2) + \Theta(n)$ ➡ $T(n) = \Theta(n \log n)$
  - $\sum_{i=1}^{n} i = \frac{1}{2} n(n+1) = \Theta(n^2)$
  - $\sum_{i=1}^{n} i^2 = \Theta(n^3)$

Sorting in linear time

The sorting problem

Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: a permutation of the input such that $\langle a_{i1} \leq \ldots \leq a_{in} \rangle$

Why do we care so much about sorting?

- sorting is used by many applications
- (first) step of many algorithms
- many techniques can be illustrated by studying sorting

Can we sort faster than $\Theta(n \log n)$??

Worst case running time of sorting algorithms:

SelectionSort:	$O(n^2)$
InsertionSort:	$O(n^2)$
MergeSort:	$O(n \log n)$

Can we do this faster? $\Theta(n \log\log n)$? $\Theta(n)$?

Upper and lower bounds

Upper bound

How do you show that a problem (for example sorting) can be solved in $\Theta(f(n))$ time?

➡ give an algorithm that solves the problem in $\Theta(f(n))$ time.

Lower bound

How do you show that a problem (for example sorting) cannot be solved faster than in $\Theta(f(n))$ time?

➡ prove that every possible algorithm that solves the problem needs $\Omega(f(n))$ time.

Lower bounds

Lower bound

How do you show that a problem (for example sorting) can not be solved faster than in $\Theta(f(n))$ time?

➡ prove that every possible algorithm that solves the problem needs $\Omega(f(n))$ time.

Model of computation: which operations is the algorithm allowed to use?

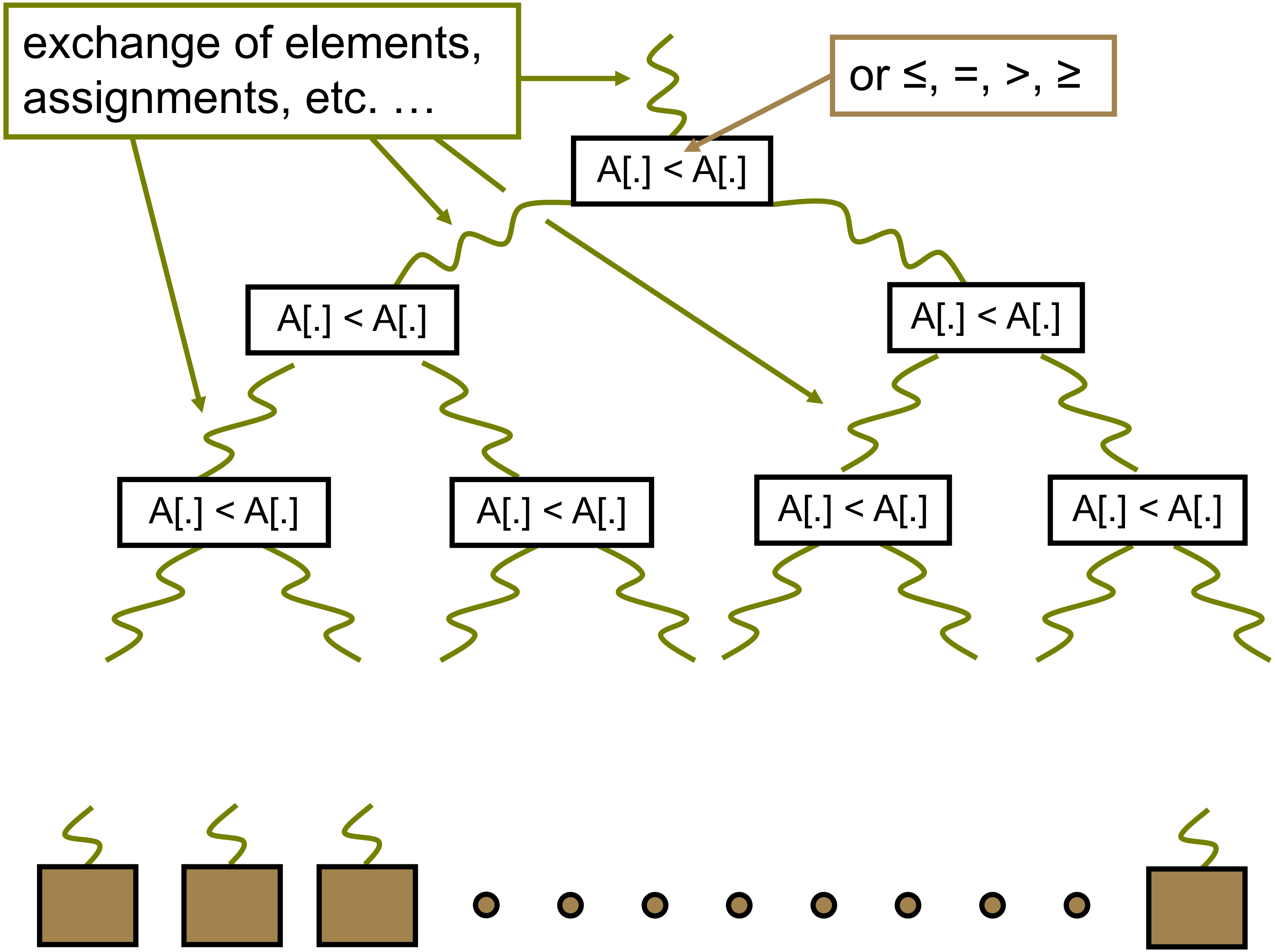
Bit-manipulations?
Random-access (array indexing) vs. pointer-machines?

Comparison-based sorting

```
SelectionSort(A, n)
    for i = 1 to n-1:
        set smallest to i
        for j = i + 1 to n:
            if A[j] < A[smallest]: set smallest to j
        swap A[i] with A[smallest]
```

Which steps precisely the algorithm executes — and hence, which element ends up where — only depends on the result of comparisons between the input elements.

Decision tree for comparison-based sorting

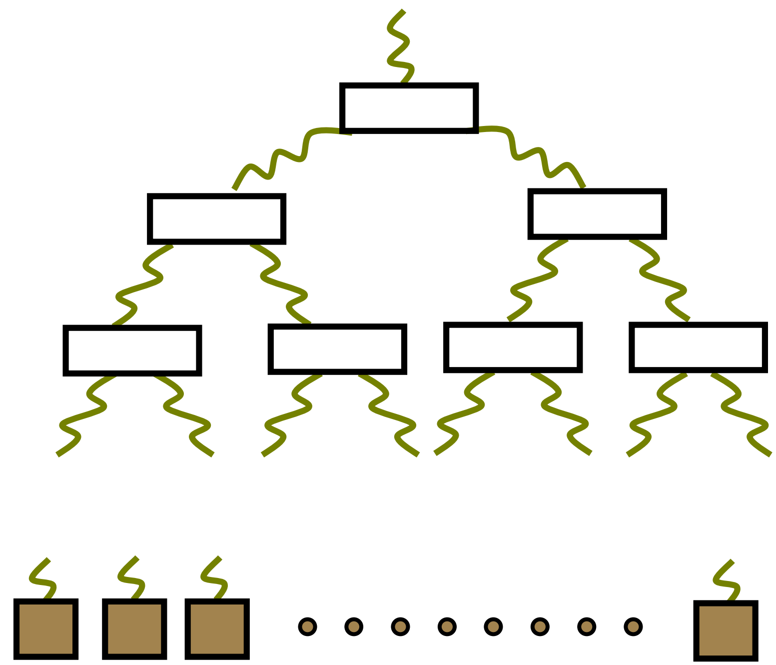


Comparison-based sorting

- every permutation of the input follows a different path in the decision tree
  ➡ the decision tree has at least $n!$ leaves

- the height of a binary tree with $n!$ leaves is at least $\log(n!)$

- worst case running time
  ≥ longest path from root to leaf
  = the height of the tree
  ≥ $\log(n!) = \Omega(n \log n)$



Lower bound for comparison-based sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

➡ The worst case running time of MergeSort is optimal.

Sorting in linear time …

Three algorithms which are faster:

1. CountingSort
2. RadixSort
3. BucketSort

(not comparison-based, make assumptions on the input)

CountingSort

Input: array A[1..n] of numbers

Assumption: the input elements are integers in the range 0 to k, for some k

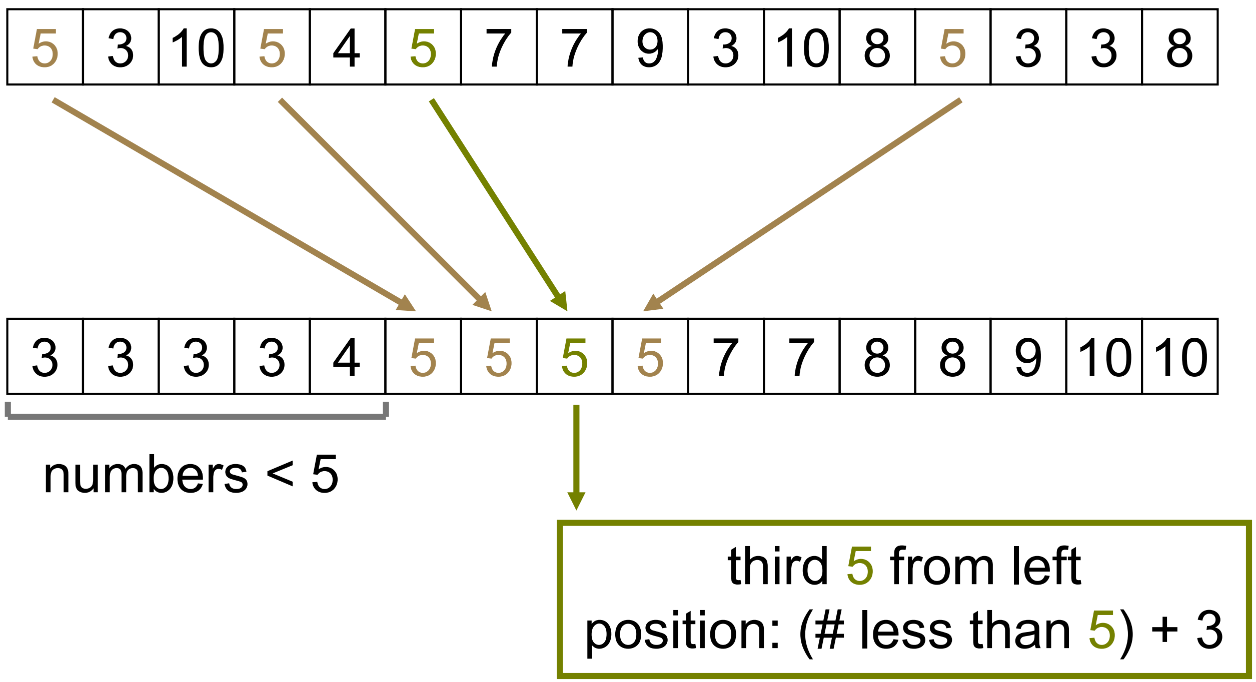
Main idea: count for every A[i] the number of elements less than A[i]
➡ position of A[i] in the output array

Beware of elements that have the same value!

position(i) = number of elements less than A[i] in A[1..n]
+ number of elements equal to A[i] in A[1..i]

CountingSort

position(i) = number of elements less than A[i] in A[1..n]
+ number of elements equal to A[i] in A[1..i]



CountingSort

position(i) = number of elements less than A[i] in A[1..n]
+ number of elements equal to A[i] in A[1..i]

Lemma

If every element A[i] is placed on position(i), then the array is sorted and the sorted order is stable.

Numbers with the same value appear in the same order in the output array as they do in the input array.

CountingSort

C[i] will contain the number of elements ≤ i

CountingSort(A,k)

► Input: array A[1..n] of integers in the range 0..k

► Output: array B[1..n] which contains the elements of A, sorted

1. **for** i = 0 **to** k **do** C[i] = 0
2. **for** j = 1 **to** A.length **do** C[A[j]] = C[A[j]] + 1
3. ► C[i] now contains the number of elements equal to i
4. **for** i = 1 **to** k **do** C[i] = C[i] + C[i-1]
5. ► C[i] now contains the number of elements less than or equal to i
6. **for** j = A.length **downto** 1
7. **do** B[C[A[j]]] = A[j]; C[A[j]] = C[A[j]] – 1

CountingSort

CountingSort(A,k)

► Input: array A[1..n] of integers in the range 0..k
► Output: array B[1..n] which contains the elements of A, sorted

1. **for** i = 0 **to** k **do** C[i] = 0
2. **for** j = 1 **to** A.length **do** C[A[j]] = C[A[j]] + 1
3. ► C[i] now contains the number of elements equal to i
4. **for** i = 1 **to** k **do** C[i] = C[i] + C[i-1]
5. ► C[i] now contains the number of elements less than or equal to i
6. **for** j = A.length **downto** 1
7. **do** B[C[A[j]]] = A[j]; C[A[j]] = C[A[j]] – 1

Correctness lines 6/7: Invariant

Inv(j): for $j + 1 \le i \le n$: B[position(i)] contains A[i]

for $0 \le i \le k$: C[i] = (# numbers smaller than i)
+ (# numbers equal to i in A[1..j])

Inv(j) holds before loop is executed, Inv(j – 1) holds afterwards

CountingSort: running time

CountingSort(A,k)

► Input: array A[1..n] of integers in the range 0..k
► Output: array B[1..n] which contains the elements of A, sorted

1. **for** i = 0 **to** k **do** C[i] = 0
2. **for** j = 1 **to** A.length **do** C[A[j]] = C[A[j]] + 1
3. ► C[i] now contains the number of elements equal to i
4. **for** i = 1 **to** k **do** C[i] = C[i] + C[i-1]
5. ► C[i] now contains the number of elements less than or equal to i
6. **for** j = A.length **downto** 1
7. **do** B[C[A[j]]] = A[j]; C[A[j]] = C[A[j]] – 1

line 1: $\sum_{0 \le i \le k} \Theta(1) = \Theta(k)$

line 2: $\sum_{1 \le i \le n} \Theta(1) = \Theta(n)$

line 4: $\sum_{0 \le i \le k} \Theta(1) = \Theta(k)$

lines 6/7: $\sum_{1 \le i \le n} \Theta(1) = \Theta(n)$

Total: $\Theta(n+k) \Rightarrow \Theta(n)$ if $k = O(n)$

CountingSort

Theorem

CountingSort is a stable sorting algorithm that sorts an array of n integers in the range $0..k$ in $\Theta(n+k)$ time.

RadixSort

Input: array A[1..n] of numbers

Assumption: the input elements are integers with d digits

example (d = 4): 3288, 1193, 9999, 0654, 7243, 4321

d^{th} digit 1^{st} digit

RadixSort(A, d)

1. **for** i = 1 **to** d
2. **do** use a stable sort to sort array A on digit i

RadixSort: example

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

sort on 1st digit → sort on 2nd digit → sort on 3rd digit

Correctness (Invariant): Before iteration i the numbers are correctly sorted on the first i-1 digits

RadixSort

Running time: If we use CountingSort as stable sorting algorithm

➡ $\Theta(n + k)$ per digit

each digit is an integer in the range 0..k

Theorem

Given n d-digit numbers in which each digit can take up to k possible values, RadixSort correctly sorts these numbers in $\Theta(d\,(n + k))$ time.

BucketSort

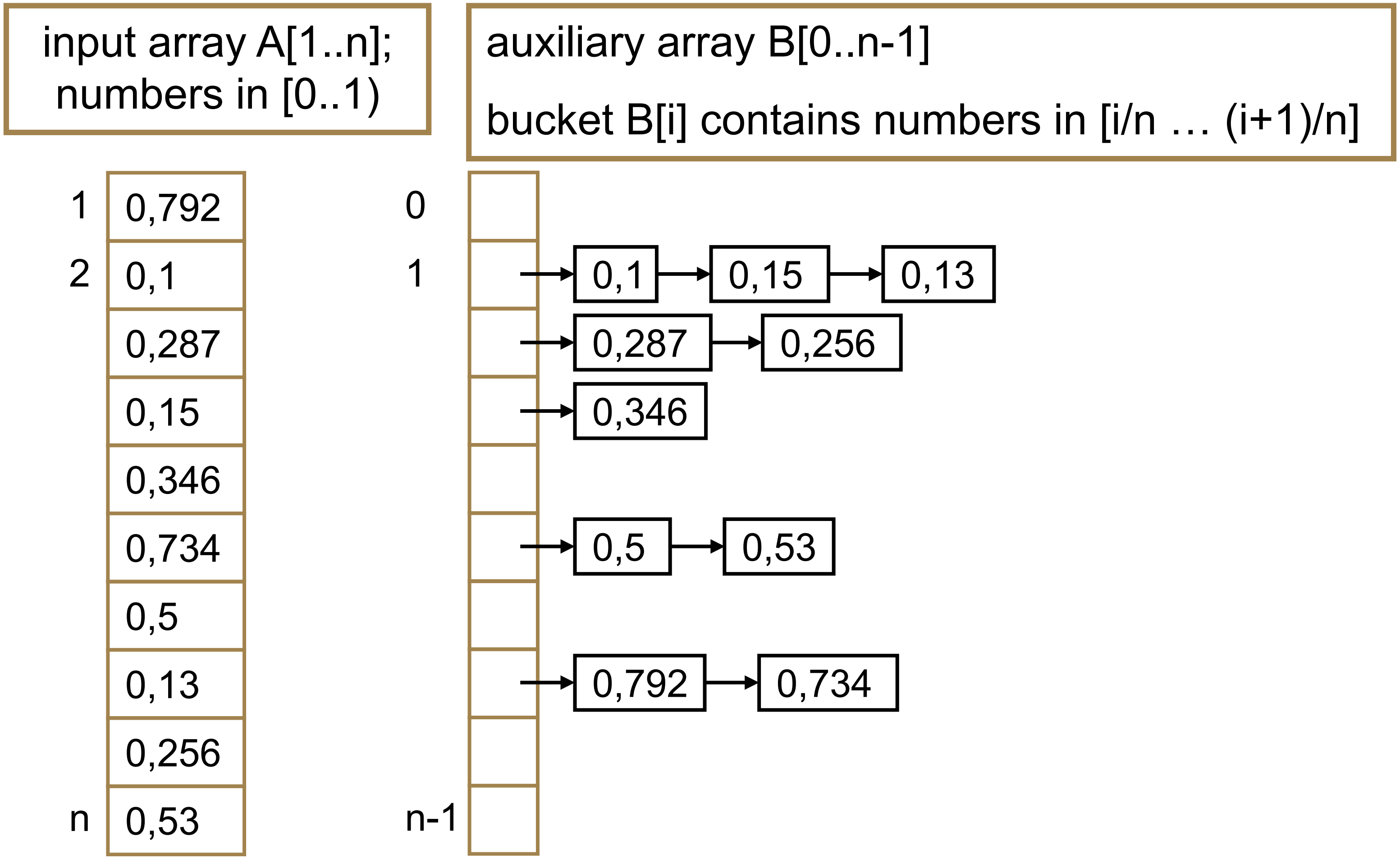
Input: array A[1..n] of numbers

Assumption: the input elements lie in the interval [0..1) (no integers!)

BucketSort is fast if the elements are uniformly distributed in [0..1)

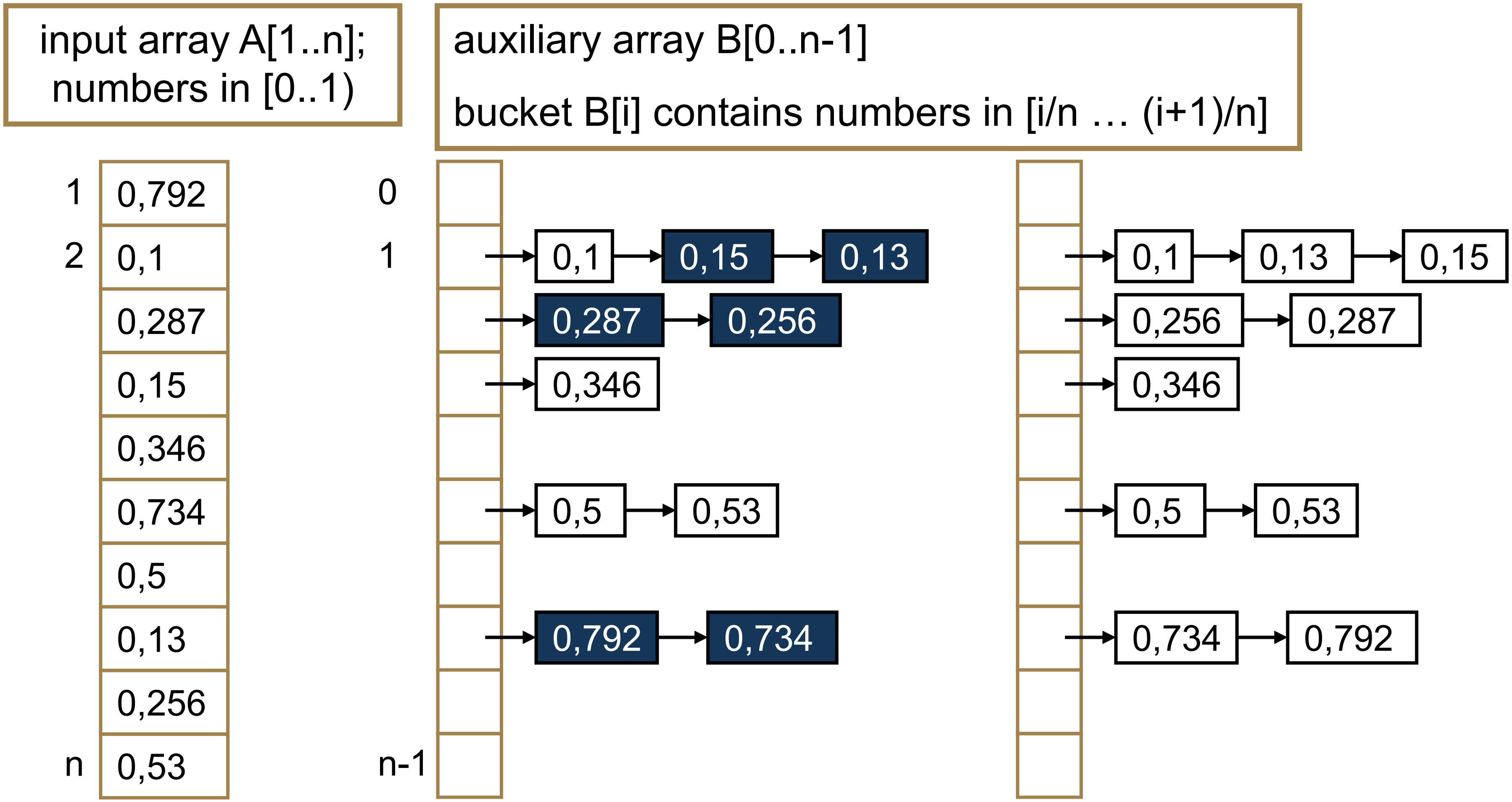
BucketSort

- Throw input elements in “buckets”, sort buckets, concatenate …



BucketSort

- Throw input elements in “buckets”, sort buckets, concatenate …



BucketSort

BucketSort(A)

► Input: array A[1..n] of numbers with $0 \le A[i] < 1$

► Output: sorted list, which contains the elements of A

n = A.length
initialize auxiliary array B[0..n-1]; each B[i] is a linked list of numbers
for i = 1 **to** n
    do insert A[i] into list B[$\lfloor n \cdot A[i] \rfloor$]
for i = 0 **to** n-1
    do sort list B[i], for example with InsertionSort
concatenate the lists B[0], B[1], …, B[n-1] together in order

BucketSort

Running time?

Define n_i = number of elements in bucket B[i]

➡ running time = $\Theta(n) + \sum_{0 \le i \le n-1} \Theta(n_i^2)$

- worst case: all numbers fall into the same bucket ➡ $\Theta(n^2)$
- best case: all numbers fall into different buckets ➡ $\Theta(n)$

- expected running time if the numbers are randomly distributed ?

BucketSort: expected running time

Define n_i = number of elements in bucket B[i]

➡ running time $= \Theta(n) + \sum_{0 \le i \le n-1} \Theta(n_i^2)$

Assumption: Pr { A[j] falls in bucket B[i] } = 1/n for each i

$$E[\text{ running time }] = E[\Theta(n) + \sum_{0 \le i \le n-1} \Theta(n_i^2)]$$
$$= \Theta(n + \sum_{0 \le i \le n-1} E[n_i^2])$$

What is $E[n_i^2]$? We have $E[n_i] = 1$ … but $E[n_i^2] \neq E[n_i]^2$

(*some math with indicator random variables – see book for details*)

➡ $E[n_i^2] = 2 - 1/n = \Theta(1)$

➡ expected running time $= \Theta(n)$

Linear time sorting

Sorting in linear time

- Only if assumptions hold!

CountingSort

- Assumption: input elements are integers in the range 0 to k
- Running time: $\Theta(n+k)$ ➡ $\Theta(n)$ if $k = O(n)$

RadixSort

- Assumption: input elements are integers with d digits
- Running time: $\Theta(d\,(n+k))$
- Can be $\Theta(n)$ for bounded integers with good choice of base

BucketSort

- Assumption: input elements lie in the interval [0..1)
- Running time: $\Theta(n)$ if elements uniformly distributed

Recap and preview

This lecture

- Models of computation and lower bounds
- Sorting is in $\Omega(n \log n)$
- Linear-time sorting under assumptions
  - Counting sort
  - Radix sort
- Partition
  - Another sorting algorithm: Quicksort
  - While sorting takes $\Omega(n \log n)$ time, median-finding takes $O(n)$ time

Next lecture

- Basic Data Structures
- Abstract Data Types