Algorithms and Datastructures

Lecture 6

Manfred Jaeger

AALBORG UNIVERSITET

Quicksort Average Case

Uniform input distribution

Assumption: for the input array A all orderings (permutations) of its elements are equally likely to occur.

☞ not always realistic, because arrays may more likely be partially sorted already

Illustration

Assumptions:
- Count only PARTITION time
- Partitioning array of length n has exactly cost n

$n = 3$:

Permutation	Time
1,2,3	3+2
1,3,2	3
2,1,3	3+2
2,3,1	3+2
3,1,2	3
3,2,1	3+2
Total	26

Average = 26/6 = 4.33

$n = 4$:

Permutation	Sum of times
,,*,4	$4 \cdot 6 + 26$
,,*,1	$4 \cdot 6 + 26$
,,*,3	$4 \cdot 6 + 2 \cdot 6$
,,*,2	$4 \cdot 6 + 2 \cdot 6$
Total	172

Average = 172/24 = 7.16

Randomized-Quicksort

- In PARTITION: first exchange $A[r]$ with a *randomly selected* element from $A[p\,..\,r]$.

Expected Runtime

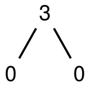
Average (expected) runtime for a *specific input* is the (probability-weighted) average taken over all possible executions for the input.

Example: possible recursion trees for input: $(1, 2, 3)$:

1 or 3 is randomly selected:

Time: 5
Probability: 2/3

2 is randomly selected:



Time: 3
Probability: 1/3

☞ Expected time = $(2/3) \cdot 5 + (1/3) \cdot 3 = 4.33$ The same for every input of length 3!

Results

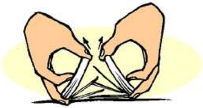
Expected runtime of RANDOMIZED-QUICKSORT for any input of size n

= Average runtime of QUICKSORT for inputs of size n under the uniform input distribution assumption

= $O(n \lg n)$

Intuition

- The uniform input distribution assumption can be made true by an initial (randomized) “shuffling” of the input



- After such an initial shuffling, further randomization in the PARTITION steps will make no difference
- RANDOMIZED-QUICKSORT is doing the shuffling on an “as needed” basis during the execution of the algorithm

Sorting: $n \lg n$ vs. n

Reminder: all sorting algorithms considered so far only perform comparison operations on the elements of the input array.

Goal

Show that any sorting algorithm that only uses comparisons must have worst-case runtime $\Omega(n \lg n)$.

Simplifying assumptions:

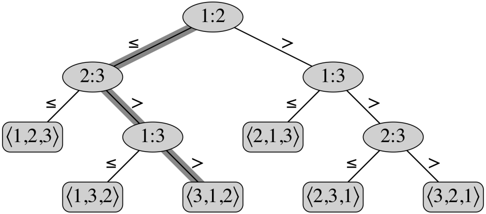
- Only use comparisons of the form $i \leq j$
- Only consider inputs with all elements distinct

Strategy: obtain lower bound by lower-bounding the number of required comparison operations

Consider a binary tree where

- inner nodes $i : j$ represent comparisons of elements $A[i]$ and $A[j]$ of the input array A
- the sub-trees of node $i : j$ contain the comparisons that will be made when $A[i] \leq A[j]$, respectively $A[i] > A[j]$.
- leaves: arrays containing the indices in the input array of the elements in the sorted output array

Decision tree for INSERTIONSORT for inputs of size 3:



Marked path: comparisons made for input $(6, 8, 5)$.

Proving the bound

The decision tree for inputs of size n of a comparison-based sorting algorithm

- must contain at least one leaf node for each of the $n!$ permutations of $1, 2, \ldots, n$.
- has height at least $\lg(n!)$.

Using that

- the height of the decision tree is a lower bound for the worst-case complexity of the algorithm
- $\lg(n!) = \Omega(n \lg n)$ (Stirling's approximation)

this shows that every comparison-based sorting algorithm has $\Omega(n \lg n)$ worst-case complexity.

Stirling's approximation: $\sqrt{2\pi n}(\frac{n}{e})^n e^{1/(12n+1)} < n! < \sqrt{2\pi n}(\frac{n}{e})^n e^{1/(12n)}$

Counting Sort

COUNTINGSORT(A)

/* $n = A.length$ */
$k = max_{i=0,\ldots,n-1} A[i]$
let $B[0..n-1]$ and $C[0..k]$ be new arrays
for $i = 0$ *to* k **do**
    C[i]=0
for $j = 0$ *to* $n-1$ **do**
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ *to* k **do**
    $C[i] = C[i] + C[i-1]$
for $j = n-1$ *to 0* **do**
    $B[C[A[j]] - 1] = A[j]$
    $C[A[j]] = C[A[j]] - 1$
return B

A	2	5	3	0	2	3	0	3

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	0	0	0	0	0
0	1	2	3	4	5

B

0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
0	0	0	0	0	0

B

0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	0	1	0	0	0
0	1	2	3	4	5

B

0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A	2	5	3	0	2	3	0	3

C	0	0	1	0	0	1
	0	1	2	3	4	5

B								
	0	1	2	3	4	5	6	7

Counting Sort

```
CountingSort(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
     C[i]=0
for j = 0 to n-1 do
     C[A[j]] = C[A[j]] + 1
for i = 1 to k do
     C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
     B[C[A[j]] - 1] = A[j]
     C[A[j]] = C[A[j]] - 1
return B
```

A	2	5	3	0	2	3	0	3

C	2	0	2	3	0	1
	0	1	2	3	4	5

B								
	0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k] be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

2	2	4	7	7	8
0	1	2	3	4	5

B

0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k] be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A	2	5	3	0	2	3	0	3

C	2	2	4	7	7	8
	0	1	2	3	4	5

B								
	0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n − 1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n − 1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i − 1]
for j = n − 1 to 0 do
    B[C[A[j]] − 1] = A[j]
    C[A[j]] = C[A[j]] − 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
2	2	4	7	7	8

B

0	1	2	3	4	5	6	7
						3	

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
2	2	4	6	7	8

B

0	1	2	3	4	5	6	7
						3	

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A	2	5	3	0	2	3	0	3

C	2	2	4	6	7	8
	0	1	2	3	4	5

B							3	
	0	1	2	3	4	5	6	7

Counting Sort

COUNTINGSORT(A)

```
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
1	2	4	6	7	8

B

0	1	2	3	4	5	6	7
	0					3	

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A	2	5	3	0	2	3	0	3

C	1	2	4	5	7	8
	0	1	2	3	4	5

B		0				3	3	
	0	1	2	3	4	5	6	7

Counting Sort

COUNTINGSORT(A)

```
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

1	2	3	5	7	8
0	1	2	3	4	5

B

	0		2		3	3	
0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	2	3	5	7	8
0	1	2	3	4	5

B

0	0		2		3	3	
0	1	2	3	4	5	6	7

Counting Sort

COUNTINGSORT(A)

/* $n = A.length$ */

$k = max_{i=0,\ldots,n-1}\ A[i]$
let $B[0..n-1]$ and $C[0..k]$ be new arrays
for $i = 0$ *to* k **do**
    C[i]=0
for $j = 0$ *to* $n-1$ **do**
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ *to* k **do**
    $C[i] = C[i] + C[i-1]$
for $j = n-1$ *to 0* **do**
    $B[C[A[j]] - 1] = A[j]$
    $C[A[j]] = C[A[j]] - 1$
return B

A	2	5	3	0	2	3	0	3

C	0	2	3	4	7	8
	0	1	2	3	4	5

B	0	0		2	3	3	3	
	0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
0	2	3	4	7	7

B

0	1	2	3	4	5	6	7
0	0		2	3	3	3	5

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k]
be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A	2	5	3	0	2	3	0	3

C	0	2	2	4	7	7
	0	1	2	3	4	5

B	0	0	2	2	3	3	3	5
	0	1	2	3	4	5	6	7

Counting Sort

```
COUNTINGSORT(A)
/* n = A.length */
k = max_{i=0,...,n-1} A[i]
let B[0..n-1] and C[0..k] be new arrays
for i = 0 to k do
    C[i]=0
for j = 0 to n-1 do
    C[A[j]] = C[A[j]] + 1
for i = 1 to k do
    C[i] = C[i] + C[i-1]
for j = n-1 to 0 do
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
return B
```

A

2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
0	2	2	4	7	7

B

0	1	2	3	4	5	6	7
0	0	2	2	3	3	3	5

Complexity: Runtime depends on n and k: $O(n + k)$.

☞ If $k = O(n)$, then $T(n) = O(n)$.

☞ Special case I: arrays of length n contain the numbers $1, \ldots, n$

☞ Special case II: all input arrays only contain values from a finite set $1, \ldots, k$, where k does not depend on n (Example: birthdates of living people – longer inputs will just contain more repetitions of the same dates)

Why linear?

- The $\Omega(n \lg n)$ lower bound for comparison-based sorting also applied for input arrays containing exactly the numbers $1, \ldots, n$.
- COUNTING SORT uses assignment operations

$$k = max_{i=0,\ldots,n-1} A[i]$$
$$C[A[j]] = C[A[j]] + 1$$
$$C[A[j]] = C[A[j]] - 1$$

that use the actual values of the input array entries.

Data Structures

A	Stack is LIFO, Queue is FIFO
B	Stack is FIFO, Queue is LIFO
C	Stack and Queue are LIFO
D	Stack and Queue are FIFO
E	This is all gibberish!

Data Structures (DS): definitions of objects/classes that

- specify how data is stored/organized
- provide methods to query and modify the data

Abstract Data Types (ADT): specifications of

- what kind of data must be stored
- what operations must be supported

	Algorithmics	OO-Programming
abstract	Abstract Data Type	Interface
implementation	Data Structure	Class
instance	(Data)	Object

Algorithms . . .

- can be *designed* and verified for *correctness* in terms of the ADTs they use
- need to be analyzed and optimized for *complexity* by considering the DSs that implement the ADTs

Dictionary of Algorithms and Data Structures at National Institute of Standards and Technology (NIST):

`http://xlinux.nist.gov/dads//HTML/abstractDataType.html`

Includes *axiomatic semantics* for ADT operations.

Many ADTs/DSs can be seen as **dynamic sets**:

Data stored is a

- set of objects, where each object may be required to
- have a designated **key** attribute, and
- the keys may further be required to come from an **ordered set**

Specific ADTs are defined in terms of specific operations for modifying and querying the set.

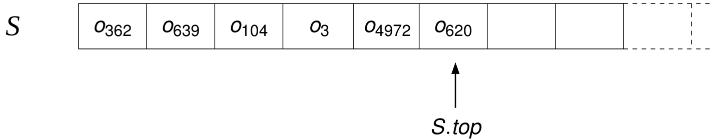
Data: set of objects

Operations:

Name	Specification
Boolean *isEmpty()*	returns *true* if the set is empty, otherwise *false*
void *push(Object o)*	adds *o* to the set
Object *pop()*	returns the object *o* that was most recently added (pushed) to the stack (if stack non-empty), and removes *o* from the stack

Objects are returned by LIFO policy: last-in-first-out.

Data stored in an array of objects with a designated index *top*:



Operations:

Name	Implementation
boolean *isEmpty()*	returns *true* if $S.top = 0$
void *push(Object o)*	assigns $S[S.top + 1] = o$, increments $S.top$ by 1
object *pop()*	returns $S[S.top]$ decrements $S.top$ by 1

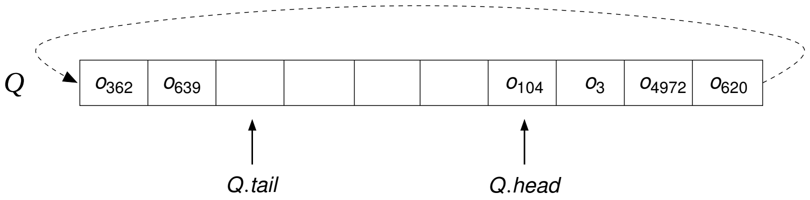
☞ all operations take time $\Theta(1)$ (regardless of the size of S)

Data: set of objects

Operations:

Name	Specification
Boolean *isEmpty()*	returns *true* if the set is empty, otherwise *false*
void *enqueue(Object o)*	adds *o* to the set
Object *dequeue()*	returns the object *o* that was least recently added (enqueued) to the queue (if queue non-empty), and removes *o* from the queue

Objects are returned by FIFO policy: first-in-first-out.



Operations:

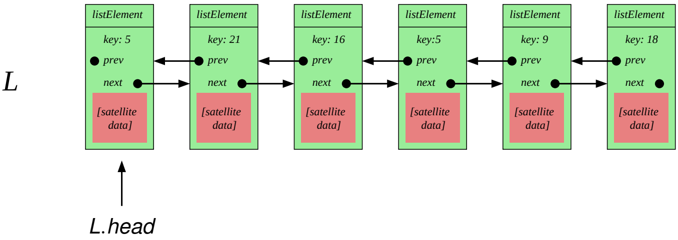
Name	Implementation
Boolean *isEmpty()*	returns *true* if $Q.head = Q.tail$
void *enqueue(Object o)*	assigns $Q[Q.tail] = o$, increments $Q.tail$ (wraps at end)
Object *dequeue()*	returns $Q[Q.head]$ increments $Q.head$ (wraps at end)

☞ all operations take time $\Theta(1)$ (regardless of the size of Q)

Data: set of objects with designated *key* attribute

Operations:

Name	Specification
Object *search(Key k)*	returns an object o with $o.key = k$ if such an object exists in the set
void *insert(Object o)*	adds o to the set
void *delete(Object o)*	deletes o from the set (o an element of the set)



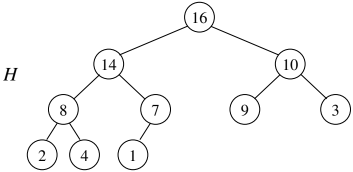
Operations:

Name	Implementation	Complexity
Object *search(Key k)*	starting at *L.head*, follows the *next* pointers until list element *o* with key *k* is found; returns *o*	$\Theta(n)$
void *insert(Object o)*	adds *o* at the start of the list (*o* becomes *L.head*)	$O(1)$
void *delete(Object o)*	redirects pointers of *o.prev* and *o.next*	$O(1)$

Data: set of objects with designated *key* attribute from an *ordered* set of keys.

Operations:

Name	Specification
void *insert(Object o)*	adds *o* to the set
Object *maximum()*	returns the object with the largest key
Object *extract-max()*	removes and returns the object with the largest key
void *increase-key(o,k)*	increases *o.key* to new value *k* (assumed to be larger than current *o.key*)



(Implemented as an array!)

Operations:

Name	Implementation	Complexity
void *insert(Object o)*	add o at $H[H.\mathit{heap\text{-}size} + 1]$; swap o with its parent until max-heap property holds	$O(\lg n)$
Object *maximum()*	return $H[1]$	$O(1)$
Object *extract-max()*	return $H[1]$; move $H[H.\mathit{heap\text{-}size}]$ to $H[1]$, decrement heap-size, call HEAPIFY(1)	$O(\lg n)$
void *increase-key(o,k)*	set $o.key = k$; swap o with its parent until max-heap property holds	$O(\lg n)$