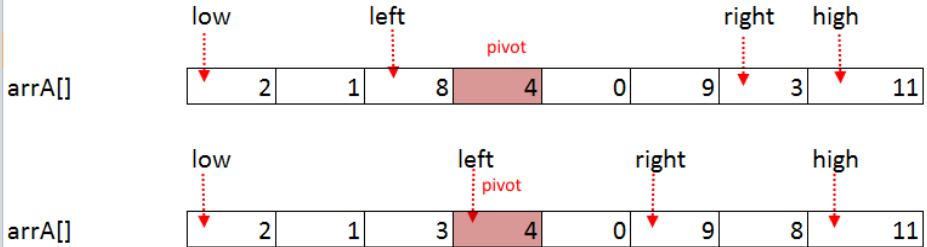


Algorithmen I - Tutorium 6

Sebastian Schmidt – *isibboi@gmail.com*

Arbeitsgruppe Kryptographie und Sicherheit



Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
 if $|s| \leq 1$ **then return** s
 pick “some” $p \in s$
 $a := \langle e \in s : e < p \rangle$
 $b := \langle e \in s : e = p \rangle$
 $c := \langle e \in s : e > p \rangle$
 return concatenation of quickSort(a), b , and quickSort(c)

Beispiel: 5, 1, 0, 2, 4, 3, 6

Pivotwahl: Rechts

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
 if $|s| \leq 1$ **then return** s
 pick “some” $p \in s$
 $a := \langle e \in s : e < p \rangle$
 $b := \langle e \in s : e = p \rangle$
 $c := \langle e \in s : e > p \rangle$
 return concatenation of quickSort(a), b , and quickSort(c)

Was ist eine Worst-Case-Eingabe, wenn als Pivot immer das rechte Element gewählt wird?

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
 if $|s| \leq 1$ **then return** s
 pick “some” $p \in s$
 $a := \langle e \in s : e < p \rangle$
 $b := \langle e \in s : e = p \rangle$
 $c := \langle e \in s : e > p \rangle$
 return concatenation of quickSort(a), b , and quickSort(c)

Was ist eine Best-Case-Eingabe mit sieben Elementen, wenn als Pivot immer das rechte Element gewählt wird?

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
 if $|s| \leq 1$ **then return** s
 pick “some” $p \in s$
 $a := \langle e \in s : e < p \rangle$
 $b := \langle e \in s : e = p \rangle$
 $c := \langle e \in s : e > p \rangle$
 return concatenation of quickSort(a), b , and quickSort(c)

Wie muss man das Pivot wählen, um eine aufsteigend sortierte Folge schnell zu sortieren?

```
Function quickSort( $s$  : Sequence of Element) : Sequence of Element  
  if  $|s| \leq 1$  then return  $s$   
  pick “some”  $p \in s$   
   $a := \langle e \in s : e < p \rangle$   
   $b := \langle e \in s : e = p \rangle$   
   $c := \langle e \in s : e > p \rangle$   
  return concatenation of quickSort( $a$ ),  $b$ , and quickSort( $c$ )
```

Wie wählt man das perfekte Pivot?
Ist das praktikabel?

```
Function quickSort( $s$  : Sequence of Element) : Sequence of Element  
  if  $|s| \leq 1$  then return  $s$   
  pick “some”  $p \in s$   
   $a := \langle e \in s : e < p \rangle$   
   $b := \langle e \in s : e = p \rangle$   
   $c := \langle e \in s : e > p \rangle$   
  return concatenation of quickSort( $a$ ),  $b$ , and quickSort( $c$ )
```

Welche Pivotwahl-Heuristik würdet ihr wählen, wenn ihr Quicksort implementieren müsstet?

- Standardfunktion aus der C++-STL
- Der vermutlich meistbenutzte Sortieralgorithmus der Welt
- Funktionsweise:
 - Startet mit halbrekursivem median of three Quicksort
 - Wenn der Quicksort eine Partitionsgröße kleiner oder gleich 16 erreicht, wird auf Insertionsort gewechselt
 - Wenn die Rekursionstiefe vom Quicksort $2 \log_2(n)$ übersteigt, wird für die entsprechende Partition auf Heapsort gewechselt
 - Der Heapsort wird ausgeführt, bis der Heap die Größe 16 hat, dann wird auf Insertionsort gewechselt

Warum startet man mit Quicksort?

- Standardfunktion aus der C++-STL
- Der vermutlich meistbenutzte Sortieralgorithmus der Welt
- Funktionsweise:
 - Startet mit halbrekursivem median of three Quicksort
 - Wenn der Quicksort eine Partitionsgröße kleiner oder gleich 16 erreicht, wird auf Insertionsort gewechselt
 - Wenn die Rekursionstiefe vom Quicksort $2 \log_2(n)$ übersteigt, wird für die entsprechende Partition auf Heapsort gewechselt
 - Der Heapsort wird ausgeführt, bis der Heap die Größe 16 hat, dann wird auf Insertionsort gewechselt

Warum startet man mit Quicksort?

- Standardfunktion aus der C++-STL
- Der vermutlich meistbenutzte Sortieralgorithmus der Welt
- Funktionsweise:
 - Startet mit halbrekursivem median of three Quicksort
 - Wenn der Quicksort eine Partitionsgröße kleiner oder gleich 16 erreicht, wird auf Insertionsort gewechselt
 - Wenn die Rekursionstiefe vom Quicksort $2 \log_2(n)$ übersteigt, wird für die entsprechende Partition auf Heapsort gewechselt
 - Der Heapsort wird ausgeführt, bis der Heap die Größe 16 hat, dann wird auf Insertionsort gewechselt

Warum startet man mit Quicksort?

- Standardfunktion aus der C++-STL
- Der vermutlich meistbenutzte Sortieralgorithmus der Welt
- Funktionsweise:
 - Startet mit halbrekursivem median of three Quicksort
 - Wenn der Quicksort eine Partitionsgröße kleiner oder gleich 16 erreicht, wird auf Insertionsort gewechselt
 - Wenn die Rekursionstiefe vom Quicksort $2 \log_2(n)$ übersteigt, wird für die entsprechende Partition auf Heapsort gewechselt
 - Der Heapsort wird ausgeführt, bis der Heap die Größe 16 hat, dann wird auf Insertionsort gewechselt

Warum benutzt man auf kleinen Partitionen Insertionsort?

- Standardfunktion aus der C++-STL
- Der vermutlich meistbenutzte Sortieralgorithmus der Welt
- Funktionsweise:
 - Startet mit halbrekursivem median of three Quicksort
 - Wenn der Quicksort eine Partitionsgröße kleiner oder gleich 16 erreicht, wird auf Insertionsort gewechselt
 - Wenn die Rekursionstiefe vom Quicksort $2 \log_2(n)$ übersteigt, wird für die entsprechende Partition auf Heapsort gewechselt
 - Der Heapsort wird ausgeführt, bis der Heap die Größe 16 hat, dann wird auf Insertionsort gewechselt

Warum benutzt man Heapsort?

Quicksort (konkret)

Function partition(a : **Array of** Element; ℓ, r, k : \mathbb{N})

$p := a[k]$

swap($a[k], a[r]$)

$i := \ell$

for $j := \ell$ **to** $r - 1$ **do**

invariant

ℓ	i	j	r
$\leq p$	$> p$?	p

if $a[j] \leq p$ **then**

swap($a[i], a[j]$)

$i++$

assert

ℓ	i	r
$\leq p$	$> p$	p

swap($a[i], a[r]$)

assert

ℓ	i	r
$\leq p$	p	$> p$

return i

Beispiel:

3, 1, 5, 6, 4, 1, 3

Pivotwahl: Rechts

Quicksort (konkret)

Function partition(a : **Array of** Element; $\ell, r, k : \mathbb{N}$)

$p := a[k]$

swap($a[k], a[r]$)

$i := \ell$

for $j := \ell$ **to** $r - 1$ **do**

invariant

ℓ	i	j	r
$\leq p$	$> p$	$?$	p

if $a[j] \leq p$ **then**

swap($a[i], a[j]$)

$i++$

assert

ℓ	i	r
$\leq p$	$> p$	p

swap($a[i], a[r]$)

assert

ℓ	i	r
$\leq p$	p	$> p$

return i

Was passiert, wenn alle Elemente gleich sind?

Quicksort (konkret)

Function partition(a : **Array of** Element; ℓ, r, k : \mathbb{N})

$p := a[k]$

swap($a[k], a[r]$)

$i := \ell$

for $j := \ell$ **to** $r - 1$ **do**

invariant

ℓ	i	j	r
$\leq p$	$> p$	$?$	p

if $a[j] \leq p$ **then**

swap($a[i], a[j]$)

$i++$

assert

ℓ	i	r
$\leq p$	$> p$	p

swap($a[i], a[r]$)

assert

ℓ	i	r
$\leq p$	p	$> p$

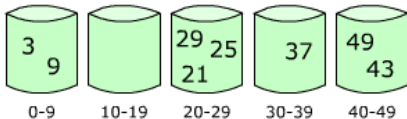
return i

Was kann man machen,
um diesen Worst-Case
zu verhindern?

Bucketsort

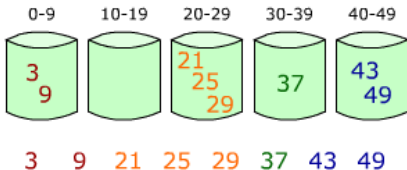
1. Elemente ihren Buckets zuordnen:

29 25 3 49 9 37 21 43



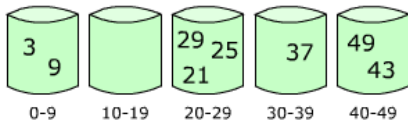
Wie sortiert man innerhalb der Buckets?

2. Innerhalb der Buckets sortieren:

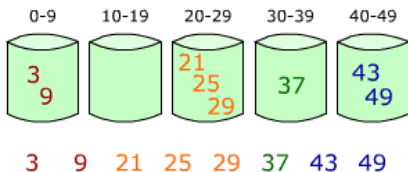


1. Elemente ihren Buckets zuordnen:

29 25 3 49 9 37 21 43



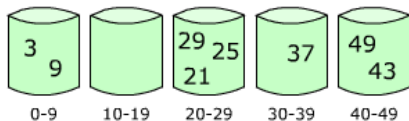
2. Innerhalb der Buckets sortieren:



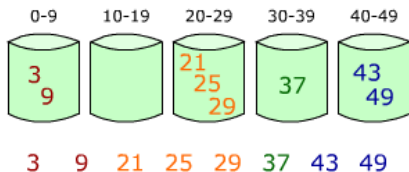
Angenommen, es gibt eine Funktion f , die jedem Element in $O(1)$ Zeit sein Bucket zuordnet. Wie ist die Best-Case-Laufzeit von Bucketsort?

1. Elemente ihren Buckets zuordnen:

29 25 3 49 9 37 21 43



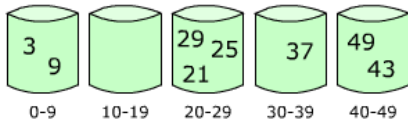
2. Innerhalb der Buckets sortieren:



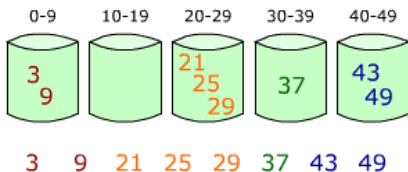
Angenommen, es gibt eine Funktion f , die jedem Element in $O(1)$ Zeit sein Bucket zuordnet. Wie ist die Worst-Case-Laufzeit von Bucketsort?

1. Elemente ihren Buckets zuordnen:

29 25 3 49 9 37 21 43



2. Innerhalb der Buckets sortieren:



Angenommen, es gibt eine Funktion f , die jedem Element in $O(1)$ Zeit sein Bucket zuordnet. Wie ist die Average-Case-Laufzeit von Bucketsort?