Datenstrukturen und Algorithmen

Exercise 3

FS 2018

Program of today

1. Feedback of last exercise
2. Repetition theory

Throwing eggs

- What would be your strategy if you would have an arbitrary number of eggs?

Throwing eggs

- What would be your strategy if you would have an arbitrary number of eggs?
  - Binary search. Worst case: $\log_2 n$ tries.

Throwing eggs

- What would be your strategy if you would have an arbitrary number of eggs?
  - Binary search. Worst case: $\log_2 n$ tries.
- What would you do if you only had one egg?

Throwing eggs

- What would be your strategy if you would have an arbitrary number of eggs?
  - Binary search. Worst case: $\log_2 n$ tries.
- What would you do if you only had one egg?
  - Start from the bottom. n tries.

Throwing Eggs

Strategy using two eggs

- First approach: intervals of equal length: partition n into k intervals: maximum number of trials

Throwing Eggs

Strategy using two eggs

- First approach: intervals of equal length: partition n into k intervals: maximum number of trials $f(k) = k + n/k - 1$
  Minimize maximum number of trials:

Throwing Eggs

Strategy using two eggs

- First approach: intervals of equal length: partition n into k intervals: maximum number of trials $f(k) = k + n/k - 1$ Minimize maximum number of trials: $f'(k) = 1 - n/k^2 = 0 \Rightarrow k = \sqrt{n}$. $n = 100 \Rightarrow 19$ Trials. $\Theta(\sqrt{n})$
- Second approach: take first throw trial into account by considering decreasing interval sizes. Choose smallest s such that $s + s - 1 + s - 2 + ... + 1 = s(s+1)/2 \geq 100 \Rightarrow s = 14$. Maximum number of trials: $s \in \Theta(\sqrt{n})$

Asymptotically both approaches are equally good. Practically the second way is better.

Selection algorithm

- What happens if many elements are equal?
- $99, 99, \dots, 99$, Pivot 99, smaller partition is empty, larger $n-1$ times 99
- May degrade runtime to n^2
- Solution?

Selection algorithm

- On equality with pivot, alternate between partitions

Selection algorithm

- On equality with pivot, alternate between partitions
- Modify algorithm to return number of elements equal to pivot

2. Repetition theory

Quiz

Consider the following three sequences of snap-shots (steps) of the algorithms (a) Insertion Sort, (b) Selection Sort and (c) Bubblesort. Below each sequence provide the corresponding algorithm name.

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

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

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

Quiz

Consider the following three sequences of snap-shots (steps) of the algorithms (a) Insertion Sort, (b) Selection Sort and (c) Bubblesort. Below each sequence provide the corresponding algorithm name.

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

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

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

Quiz

Consider the following three sequences of snap-shots (steps) of the algorithms (a) Insertion Sort, (b) Selection Sort and (c) Bubblesort. Below each sequence provide the corresponding algorithm name.

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

selection

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

bubblesort

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

insertion

Quiz

Execute two further iterations of the algorithm Quicksort on the following array. The first element of the (sub-)array serves as the pivot.

8	7	10	15	3	6	9	5	2	13
2	7	5	6	3	**8**	9	15	10	13

Quiz

Execute two further iterations of the algorithm Quicksort on the following array. The first element of the (sub-)array serves as the pivot.

8	7	10	15	3	6	9	5	2	13
2	7	5	6	3	**8**	9	15	10	13

Quiz

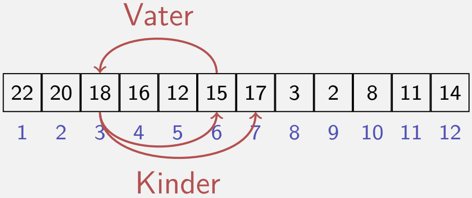
Execute two further iterations of the algorithm Quicksort on the following array. The first element of the (sub-)array serves as the pivot.

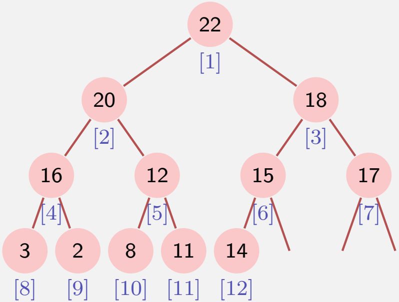
8	7	10	15	3	6	9	5	2	13
2	7	5	6	3	**8**	9	15	10	13
2	7	5	6	3	**8**	**9**	15	10	13
2	3	5	6	**7**	**8**	**9**	13	10	**15**

Heap and Array

Tree $\to$ Array:

- children$(i) = \{2i, 2i+1\}$
- parent$(i) = \lfloor i/2 \rfloor$





Depends on the starting index[1]

[1] For array that start at 0: $\{2i, 2i+1\} \to \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \to \lfloor (i-1)/2 \rfloor$

Algorithm SiftDown(A, i, m)

Input: Array A with heap structure for the children of i. Last element m.
Output: Array A with heap structure for i with last element m.

while $2i \leq m$ **do**
  $j \leftarrow 2i$; // j left child
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j + 1$; // j right child with greater key
  if $A[i] < A[j]$ **then**
    swap($A[i], A[j]$)
    $i \leftarrow j$; // keep sinking
  else
    $i \leftarrow m$; // sinking finished

Algorithm HeapSort(A, n)

Input: Array A with length n.
Output: A sorted.

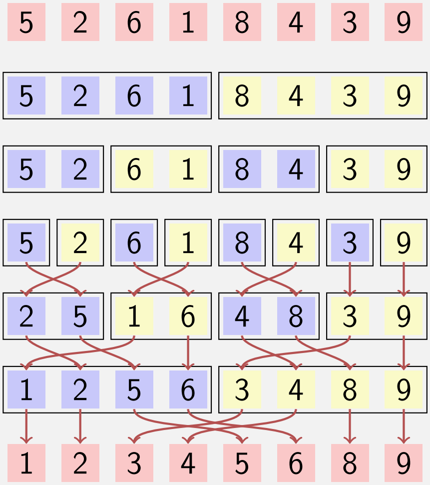
for $i \leftarrow n/2$ **downto** 1 **do**
    SiftDown(A, i, n);

// Now A is a heap.

for $i \leftarrow n$ **downto** 2 **do**
    swap($A[1], A[i]$)
    SiftDown($A, 1, i-1$)

// Now A is sorted.

Mergesort



Split

Split

Split

Merge

Merge

Merge

Algorithm recursive 2-way Mergesort(A, l, r)

Input: Array A with length n. $1 \leq l \leq r \leq n$
Output: Array $A[l, \ldots, r]$ sorted.

if $l < r$ **then**
    $m \leftarrow \lfloor (l+r)/2 \rfloor$ // middle position
    Mergesort(A, l, m) // sort lower half
    Mergesort($A, m+1, r$) // sort higher half
    Merge(A, l, m, r) // Merge subsequences

Algorithm NaturalMergesort(A)

Input: Array A with length $n > 0$
Output: Array A sorted

repeat
  $r \leftarrow 0$
  while $r < n$ **do**
    $l \leftarrow r + 1$
    $m \leftarrow l$; **while** $m < n$ **and** $A[m+1] \geq A[m]$ **do** $m \leftarrow m + 1$
    if $m < n$ **then**
      $r \leftarrow m + 1$; **while** $r < n$ **and** $A[r+1] \geq A[r]$ **do** $r \leftarrow r + 1$
      Merge(A, l, m, r);
    else
      $r \leftarrow n$
until $l = 1$

Quicksort (arbitrary pivot)

2	4	5	6	8	$\boxed{3}$	7	9	1
$\boxed{2}$	1	3	6	8	$\boxed{5}$	7	9	4
1	2	3	4	5	8	$\boxed{7}$	9	6
1	2	3	4	5	6	7	9	$\boxed{8}$
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Algorithm Quicksort($A[l, \ldots, r]$)

Input: Array A with length n. $1 \leq l \leq r \leq n$.
Output: Array A, sorted between l and r.
if $l < r$ **then**
  Choose pivot $p \in A[l, \ldots, r]$
  $k \leftarrow$ Partition($A[l, \ldots, r], p$)
  Quicksort($A[l, \ldots, k-1]$)
  Quicksort($A[k+1, \ldots, r]$)

Quicksort with logarithmic memory consumption

Input: Array A with length n. $1 \leq l \leq r \leq n$.
Output: Array A, sorted between l and r.

while $l < r$ **do**
    Choose pivot $p \in A[l, \dots, r]$
    $k \leftarrow$ Partition$(A[l, \dots, r], p)$
    if $k - l < r - k$ **then**
        Quicksort$(A[l, \dots, k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1, \dots, r])$
        $r \leftarrow k - 1$

The call of Quicksort$(A[l, \dots, r])$ in the original algorithm has moved to iteration (tail recursion!): the if-statement became a while-statement.

Stable and in-situ sorting algorithms

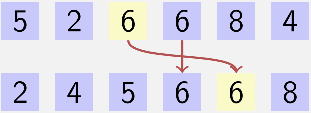
- Stable sorting algorithms don't change the relative position of two elements.

5	2	6	6	8	4
2	4	5	6	6	8

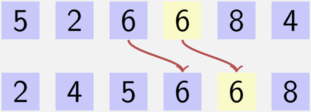
not stable

Stable and in-situ sorting algorithms

- Stable sorting algorithms don't change the relative position of two elements.



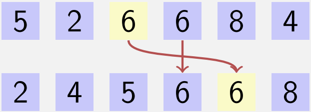
not stable



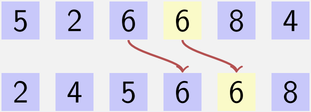
stable

Stable and in-situ sorting algorithms

- Stable sorting algorithms don't change the relative position of two elements.



not stable



stable

- In-situ algorithms require only a constant amount of additional memory.

  Which of the sorting algorithms are stable? Which are in-situ? (How) can we make them stable / in-situ?

Questions?