Föreläsning 18

Sorting and selection

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
25 November 2024

IDA, Linköpings universitet

18.1

Content

Contents

18.2

1 Sorting

1.1 Introduction

Sorting

Input:
- A list L containing data with *keys* from totally ordered set K

Output:
- A list L' with the same data sorted in increasing order wrt. the keys

Example
$[8,2,9,4,6,10,1,4] \rightarrow [1,2,4,4,6,8,9,10]$

18.3

Sorting categories

- in-place vs out-of-place sorting
- internal vs external sorting
- stable vs non-stable sorting
- Comparison vs non-comparison sorting

18.4

Strategies

Sorting by insertion
Look for the right place to insert each new element that needs to be added to the sorted sequence... *Insertion sort*, Shell sort, ...

Sorting by selection
Look, at each iteration, in the unsorted sequence for the least element left and add it to the end of sorted sequence... *Selection sort*, *Heap sort*, ...

Sorting by permutation
Search in some pattern and permute places each time a pair is found to violate the targeted order... *Quick sort*, *Merge sort*, ...

18.5

1.2 Insertion sort

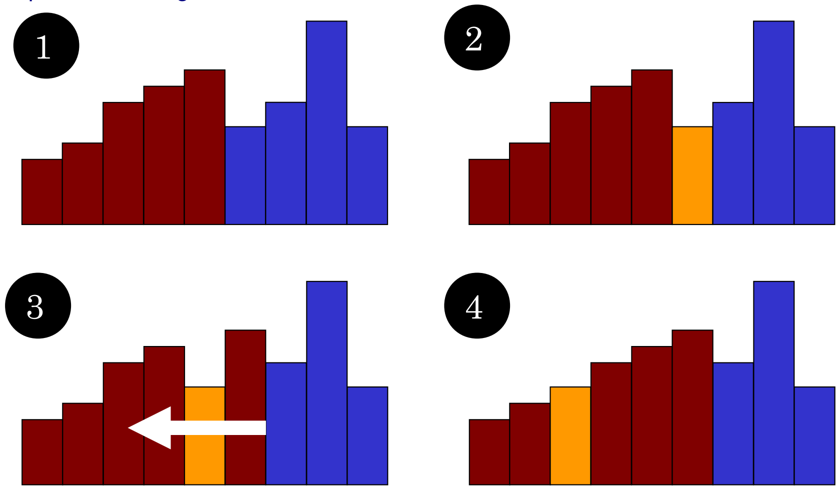
(Linear) insertion sort

- An in-place algorithm!
- Partition the array that is to be sorted $A[0,\dots,n-1]$ into two parts:
  - A sorted $A[0,\dots,i-1]$ part
  - An unsorted $A[i,\dots,n-1]$ part

  Initially, $i = 1$ and $A[0,\dots,0]$ is (trivially) sorted

procedure INSERTIONSORT($A[0,\dots,n-1]$)
    for $i = 1$ **to** $n-1$ **do**
        insert in $A[i]$ in the right position in $A[0,\dots,i-1]$ 18.6

Example: Visualizing insertion sort



18.7

Worst case analysis for insertion sort

1: **procedure** INSERTIONSORT($A[0,\dots,n-1]$)
2:     **for** $i = 1$ **to** $n-1$ **do**
3:         $j \leftarrow i$; $x \leftarrow A[i]$
4:         **while** $j \geq 1$ **and** $A[j-1] > x$ **do**
5:             $A[j] \leftarrow A[j-1]$; $j \leftarrow j-1$
6:         $A[j] \leftarrow x$

- t_2: $n-1$ times
- t_3: $n-1$ times
- t_4: Let I be the number of iterations in the worst case for the inner loop:

$$I = 1+2+\dots+(n-1) = n(n-1)/2 = (n^2-n)/2$$

- t_5: I time
- t_6: $n-1$ time
- Total: $t_2+t_3+t_4+t_5+t_6 = 3(n-1)+(n^2-n) = n^2+2n-3$ So $O(n^2)$ in the worst case… *but good if the sequence is almost sorted* 18.8

1.3 Selection sort

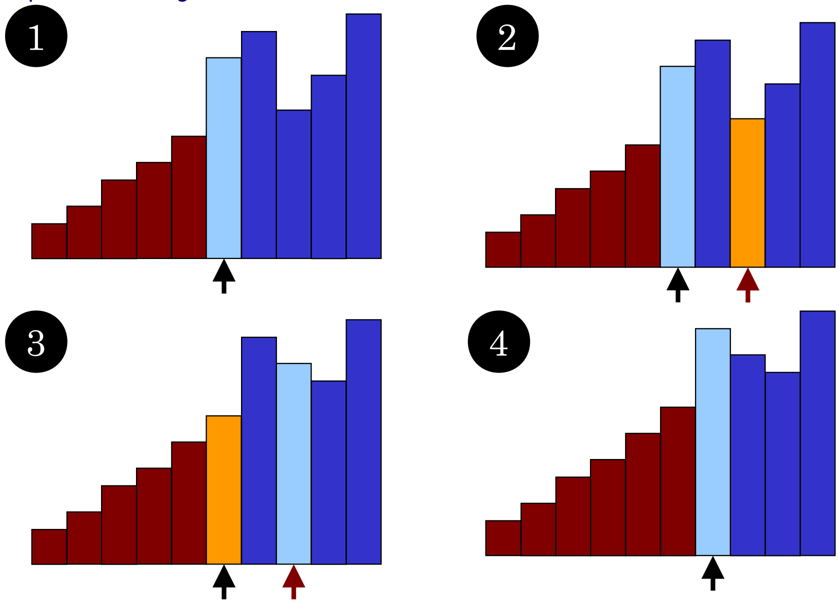
Selection sort

- An in-place algorithm
- Partition the array to be sorted $A[0,\dots,n-1]$ into two parts
  - A sorted $A[0,\dots,i-1]$ where all elements are smaller or equal to $A[i,\dots,n-1]$
  - An unsorted sequence $A[i,\dots,n-1]$

  Initially $i = 0$, hence, the sorted part is empty (and hence trivially sorted)

procedure SELECTIONSORT($A[0,\dots,n-1]$)
    for $i = 0$ **to** $n-2$ **do**
        find minimal element $A[j]$ in $A[i,\dots,n-1]$
        swap $A[i]$ and $A[j]$ 18.9

Example: Visualizing Selection-sort



18.10

Worst case analysis of Selection-sort

procedure SELECTIONSORT($A[0,\ldots,n-1]$)
  for $i = 0$ **to** $n-2$ **do**
    $s \leftarrow i$
    for $j \geq i+1$ **to** $n-1$ **do**
      if $A[j] < A[s]$ **then** $s \leftarrow j$
    SWAP($A[i], A[s]$)

- t_2: $n-1$ times
- t_3: $n-1$ times
- t_4: Let I be the number of iterations of the inner loop in the worst case:

$$I = (n-2)+(n-3)+\ldots+1 = (n-1)(n-2)/2 = (n^2-3n+2)/2$$

- t_5: I times
- t_6: $n-1$ times
- Total: $t_2+t_3+t_4+t_5+t_6 = 3(n-1)+(n^2-3n+2) = n^2-1 \in O(n^2)$

18.11

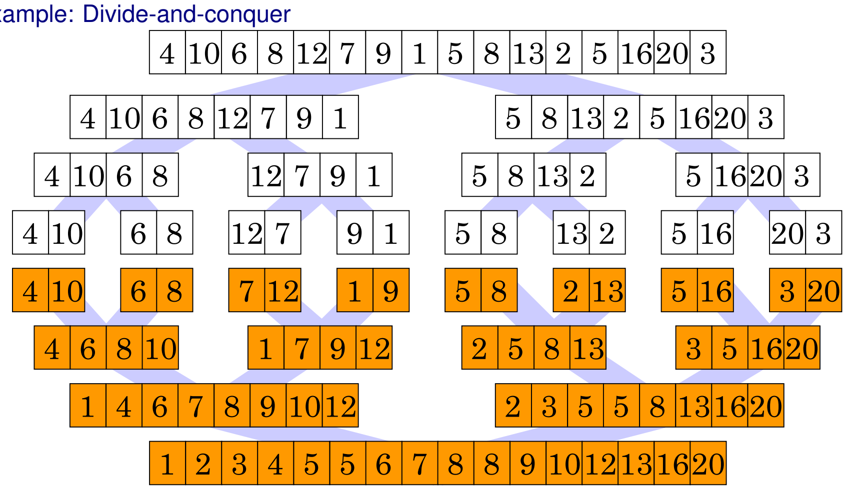
1.4 Divide-and-conquer

The divide-and-conquer paradigm in algorithm construction

- divide: divide the problem in smaller independent problems
- conquer: solve the sub-problems recursively (or directly if trivially)
- combine solutions of the sub-problems into a solution to the original problem

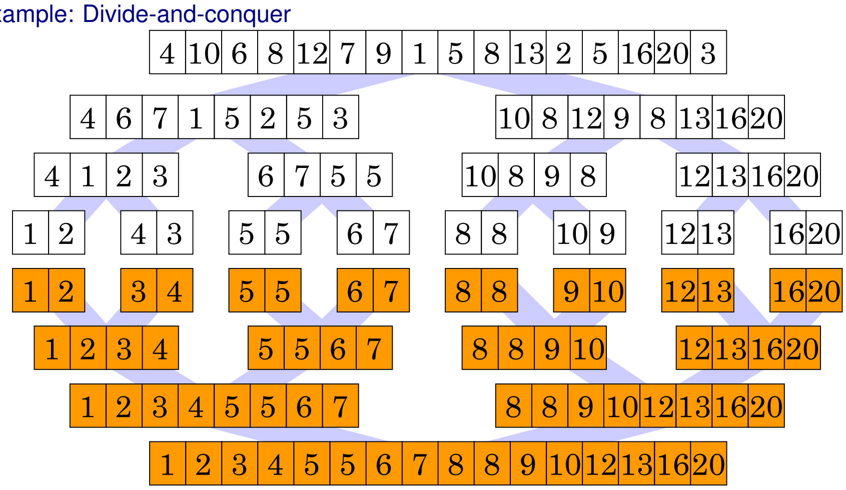
18.12

Example: Divide-and-conquer



18.13

Example: Divide-and-conquer



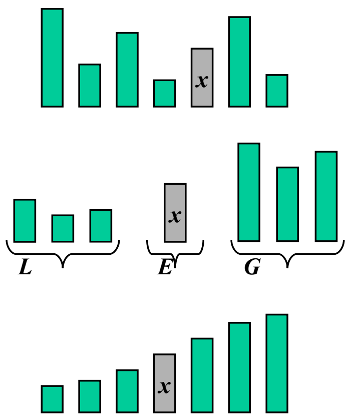
18.14

1.5 Quick-sort

Quick-sort

Quick-sort is a *randomized* sorting algorithm based on the divide-and-conquer paradigm

- Divide: randomly choose an element x (called pivot) and partition S to
  - L elements smaller than x
  - E elements equal to x
  - G elements larger than x
- Conquer: sort L and G
- Combine L, E and G



18.15

Partitioning

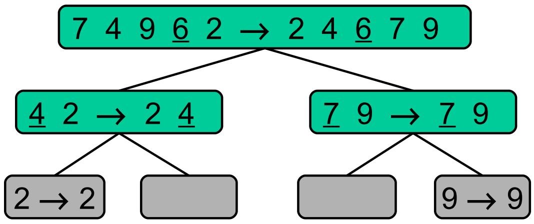
- Partition input sequence S as follows:
  - We remove, one element at a time, each element y from S and
  - Insert y in L, E or G depending on the result of the comparison with the pivot-element x
- Each insertion or removal takes place at the beginning or end of a sequence and requires $O(1)$ time
- The partitioning step takes $O(n)$ time

```
function PARTITION(S, p)
    L, E, G ← empty sequences
    x ← S.REMOVE(p)
    while ¬S.ISEMPTY() do
        y ← S.REMOVE(S.FIRST())
        if y < x then
            L.INSERTLAST(y)
        else if y = x then
            E.INSERTLAST(y)
        else
            G.INSERTLAST(y)
    return L, E, G
```

18.16

Quick-sort tree

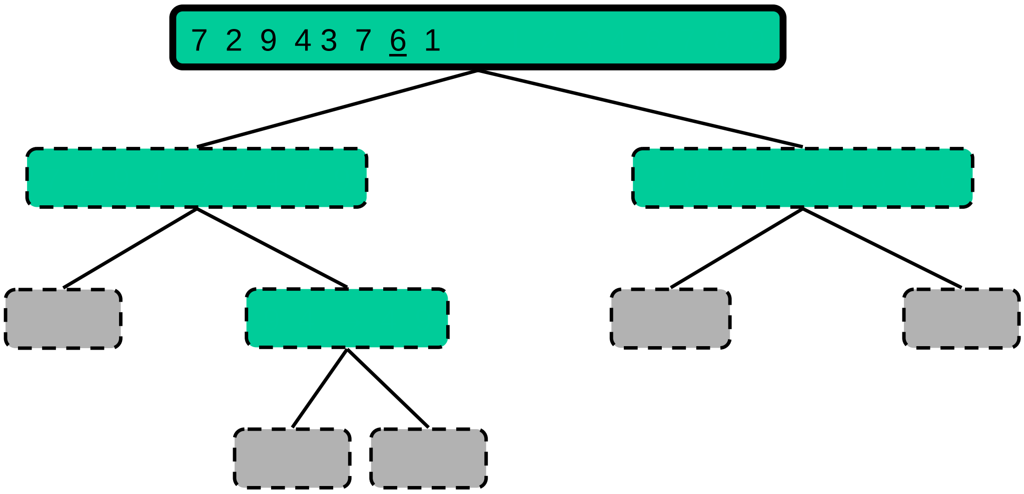
- Execution of quick-sort can be visualized as a binary tree
  - Each node represents a recursive call to quick-sort and stores
    - Unsorted sequence before execution and the pivot
    - Sorted sequence after execution
  - Root is the original call
  - Leaves are calls to sub-sequences of lengths 0 or 1



18.17

Example: Execution of quick-sort

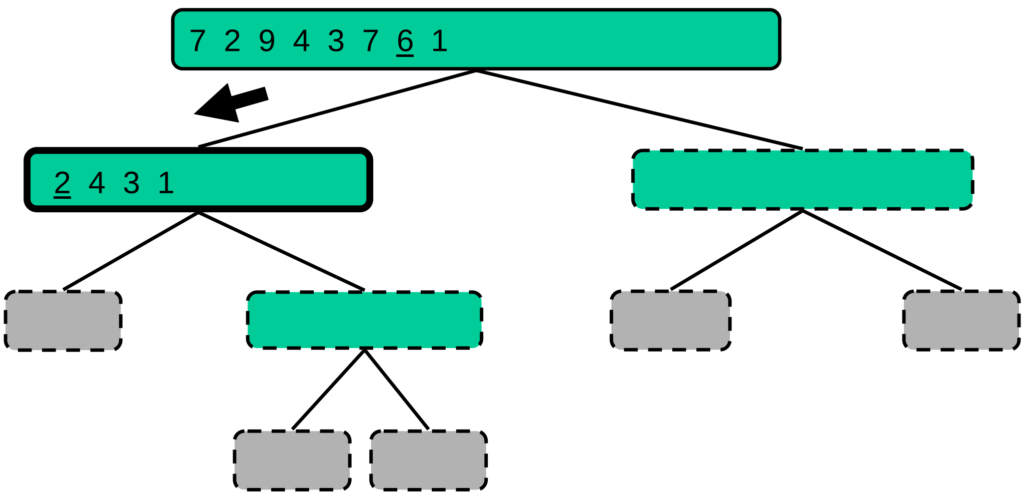
- Choice of a pivot
- example: 6, 2, 3, 7



18.18

Example: Execution of quick-sort

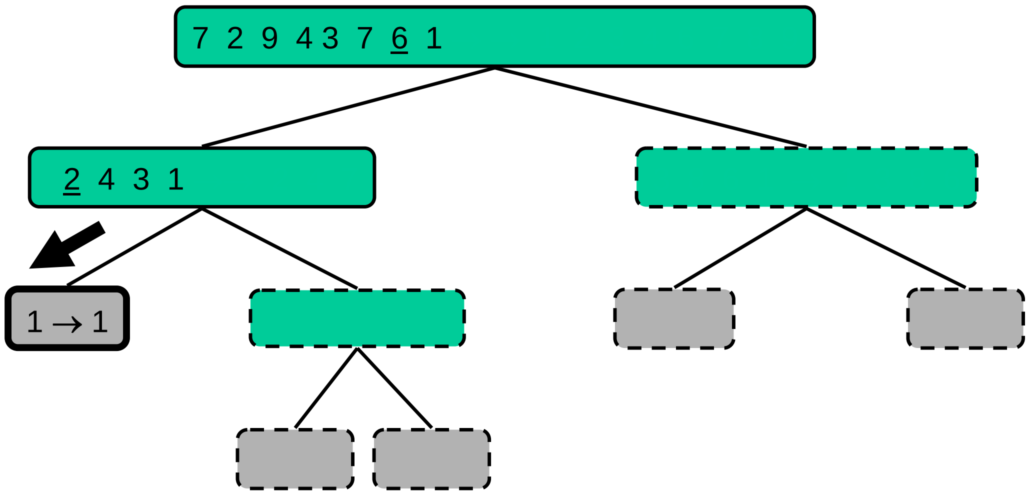
- Partitioning, recursive call, choice of a pivot



18.19

Example: Execution of quick-sort

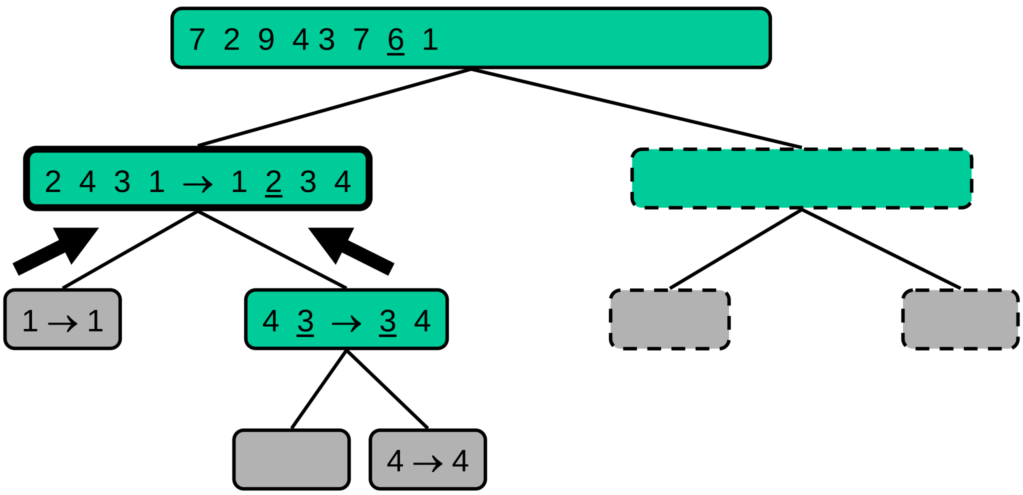
- Partitioning, recursive call, base case



18.20

Example: Execution of quick-sort

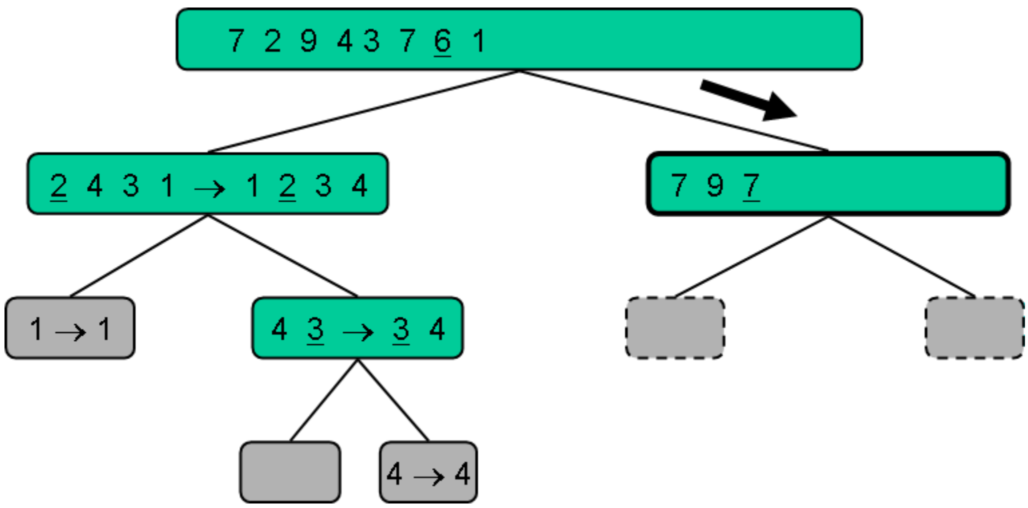
- Recursive call, . . . , base case, combine



18.21

Example: Execution of quick-sort

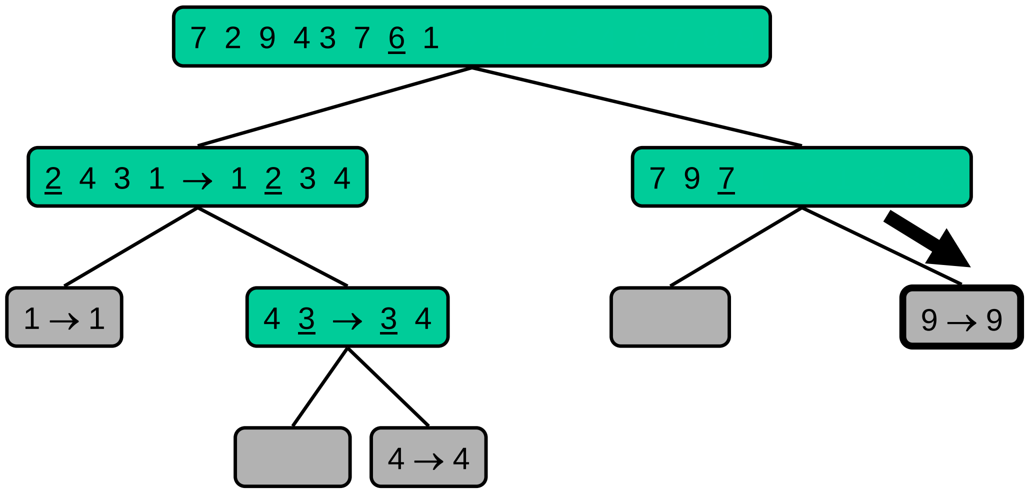
- Recursive call, choice of pivot



18.22

Example: Execution of quick-sort

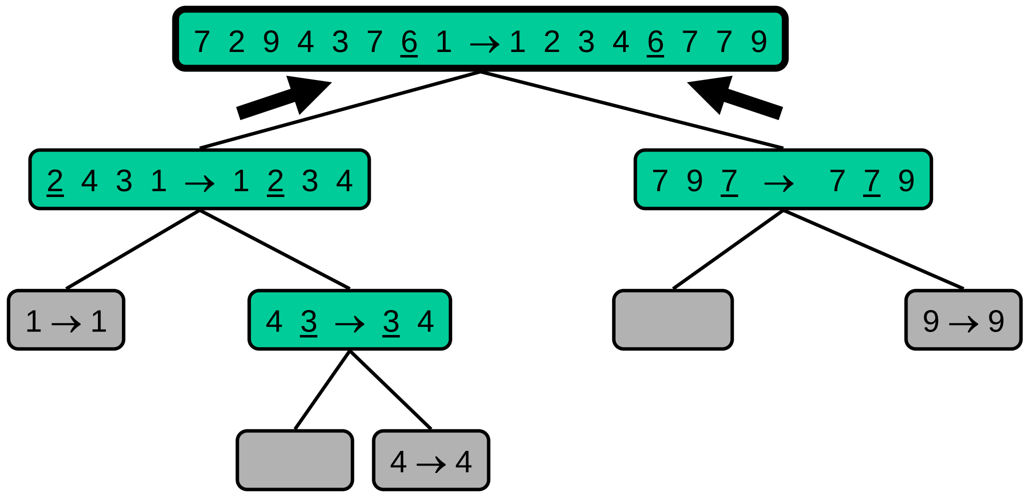
- Partitioning, ..., recursive call, base case



18.23

Example: Execution of quick-sort

- Combine



18.24

Execution time in worst-case

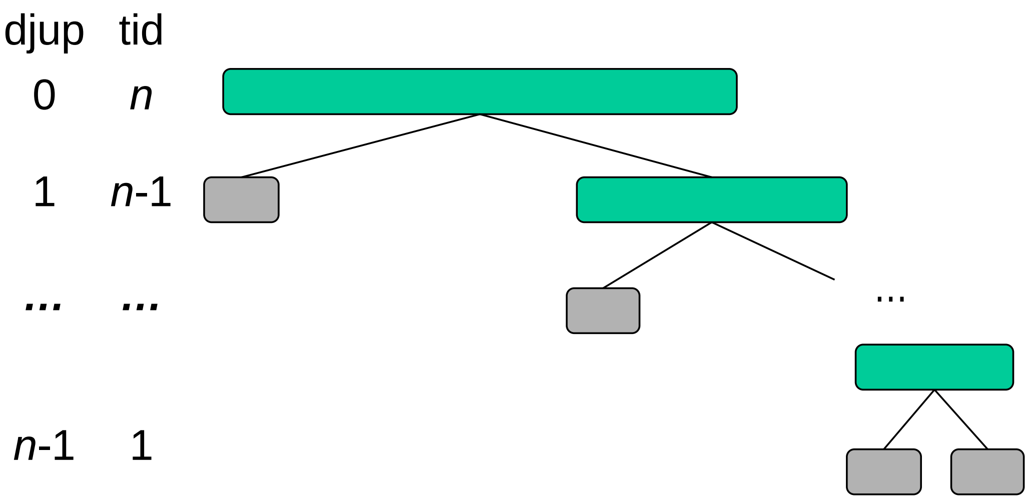
- Worst case for quick-sort happens when the pivot element is a unique minimal or maximal element
- One of L or G is $n-1$ elements long and the other is of length 0
- Execution time is then proportional to the sum

$$n+(n-1)+\ldots+2+1$$

- The worst case execution time of quick-sort is then $O(n^2)$

18.25

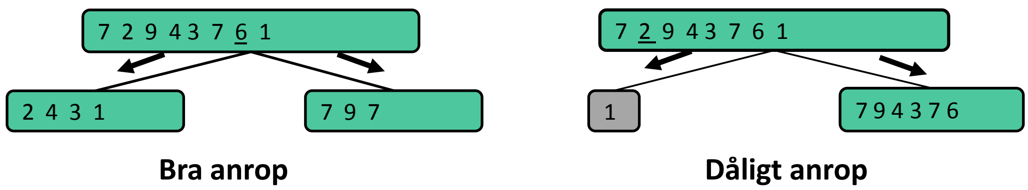
Execution time in worst-case



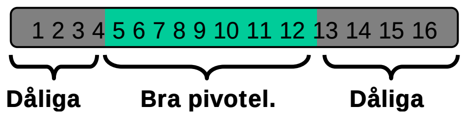
18.26

Expected execution time

- Consider a recursive call in quick-sort on a sequence of length s
  - A good call: length of L and G are both $< 3s/4$
  - A bad call: one of L or G has length $\geq 3s/4$



- A call is good with probability 1/2
  - Half of all possible pivots result in a good call:



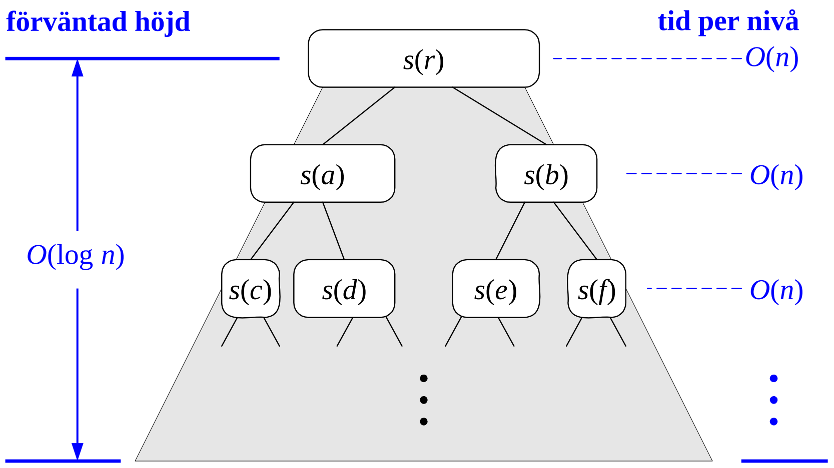
18.27

Expected execution time

- Probabilistic fact: Expected number of coin flips to obtain k heads is $2k$
- For a node at depth i, we expect:
  - $i/2$ ancestors are good calls
  - length of the input sequence is at most $(3/4)^{i/2}n$
- Consequently:
  - For a node at depth $2\log_{4/3} n$, the expected input sequence length is at most 1
  - The expected height of the quick-sort tree is $O(\log n)$
- Amount of work performed at the same depth is $O(n)$
- Therefore, the expected execution time for quick sort is $O(n \log n)$

18.28

Expected execution time

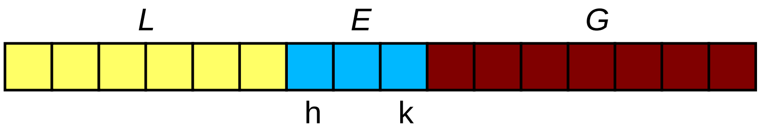


förväntad total tid: $O(n \log n)$

18.29

Quick-sort with constant extra memory

- Quick-sort can be implemented as an *in-place* sorting algorithm
- In the partitioning step, we rearrange the elements in the input sequence so that:
  - elements that are less than the pivot element have a rank that is smaller than h
  - elements that are equal to the pivot element have a rank between h and k
  - elements that are larger than the pivot element have a rank that is larger than k



18.30

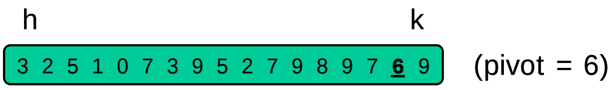
Algorithm for quick-sort with constant extra memory

procedure INPLACEQUICKSORT(S, l, r)
  if $l \geq r$ **then return**
  $i \leftarrow$ randomly chosen rank between l and r
  $x \leftarrow S$.ELEMATRANK(i)
  $(h, k) \leftarrow$INPLACEPARTITION(x)
  INPLACEQUICKSORT$(S, l, h-1)$
  INPLACEQUICKSORT$(S, k+1, r)$

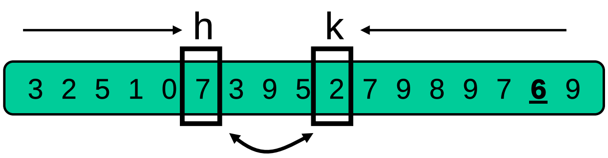
18.31

Partitioning with constant extra memory

- Perform partitioning using two indices to partition S in L and $E \cup G$ (a similar method can be used to partition $E \cup G$ in E and G)



- Repeat until h and k cross each other (i.e., $h > k$):
  - Sweep h to the right until an element $\geq$ than the pivot element is found
  - Sweep k to the left until an element $<$ pivot element is found
  - Swap the elements at indices h and k



18.32

2 Selection

2.1 Introduction

Selection problem

- Given an integer i and n elements $x_1, x_2, \ldots, x_n$ from a total order, find the ith smallest element in the sequence.

- We could sort the sequence in $O(n \log n)$ time and then index the i:th element in constant time.

i=3   7 4 9 6 2 → 2 4 6 7 9

- Can we solve the selection problem faster?

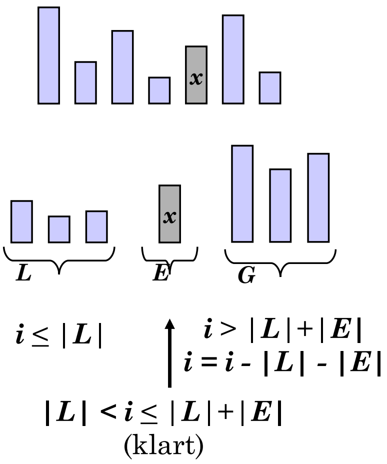
18.33

2.2 Quick-select

Quick-select

Quick-Select is a randomized selection algorithm based on the *prune-and-search* paradigm:

- *Prune*: choose x randomly and partition S into
  - L elements smaller than x
  - E elements equal to x
  - G elements larger than x
- *Search*: depending on i, the solution is either in E or we need to continue recursively in L or G



18.34

Partitioning

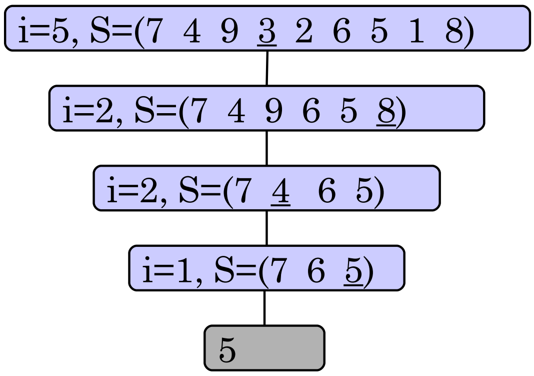
- Partition input sequence as in quick-sort:
  - We pick, one element at a time, each element y from S and
  - Insert y in L, E or G depending on the result of comparison with the pivot x
- Each insertion or removal is in the beginning or the end of a sequence, and therefore takes $O(1)$ time
- Consequently, the partitioning step in quick-select takes $O(n)$ time

```
function PARTITION(S, p)
    L, E, G ← empty sequences
    x ← S.REMOVE(p)
    while ¬S.ISEMPTY() do
        y ← S.REMOVE(S.FIRST())
        if y < x then
            L.INSERTLAST(y)
        else if y = x then
            E.INSERTLAST(y)
        else
            G.INSERTLAST(y)
    return L, E, G
```

18.35

Visualizing Quick-select

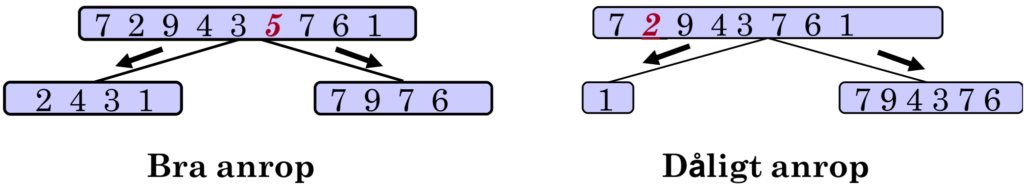
- Execution of quick-select can be visualized with the recursion path
  - Each node represents a recursive call to quick-select and stores i and the remaining sequence S



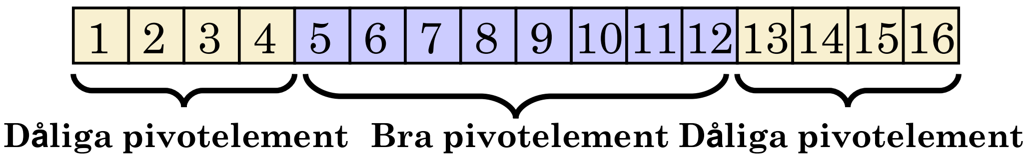
18.36

Expected execution time

- Consider a recursive call to quick-select on a sequence of length s
  - Good call: lengths of L and G are both $< 3s/4$
  - Bad call: one of L or G has a length $\geq 3s/4$



- A call is good with probability 1/2
  - Half of all the possible pivot elements result in good calls:



18.37

Expected execution time

- Probabilistic fact: The expected number of coin flips to get a head is two.
- Probabilistic fact: Expected function is linear
  - $E(X+Y) = E(X) + E(Y)$
  - $E(c \times X) = c \times E(X)$ for any constant c
- Let $T(n)$ be the expected execution time for quick-select
- We have $T(n) \leq b \cdot n \cdot g(n) + T(3n/4)$ where:
  - b is some constant
  - $g(n)$ is the expected number of call before a good call

18.38

Expected execution time

- Consequently:
  - $T(n) \leq b \cdot n \cdot g(n) + T(3n/4)$
- Since a good call is expected to happen in two steps:
  - $T(n) \leq 2 \cdot b \cdot n + T(3n/4)$
- Hence, $T(n)$ is bounded by a geometric series:
  - $T(n) \leq 2 \cdot b \cdot n + 2 \cdot b \cdot n \cdot (3/4) + 2 \cdot b \cdot n \cdot (3/4)^2 + 2 \cdot b \cdot n \cdot (3/4)^3 + \ldots$
- As a result, $T(n) \in O(n)$
- We can solve the selection problem in $O(n)$ expected time (worst case is $O(n^2)$ time)

18.39