Lecture 20

Sorting and selection

TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
22 november 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

20.1

Content

Innehåll

20.2

1 Sorting

1.1 Introduction

Sorting problem

Input:

- A list L containing data with *keys* from a linearly ordered set K

Output:

- A list L' containing the same data sorted in ascending order of keys

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

20.3

Aspects of sorting

- in-place vs use extra memory
- internal vs external memory
- stable vs non stable
- comparison-based vs digital

20.4

Strategies

Sorting through insertion
Look for the right place to insert each new element to be added in the sorted sequence... *linear insertion*, Shell-sort, ...

Sorting by selection
Search in each iteration on the unsorted sequence for the smallest remaining data and add it to the end of the sorted sequence ... *straight selection*, *Heap-sort*, ...

Sorting through location changes
Search back and forth in any pattern and swap the locations of the pair in the wrong internal order as soon as one is detected... *Quick-sort*, *Merge-sort*, ...

20.5

1.2 Insertion sort

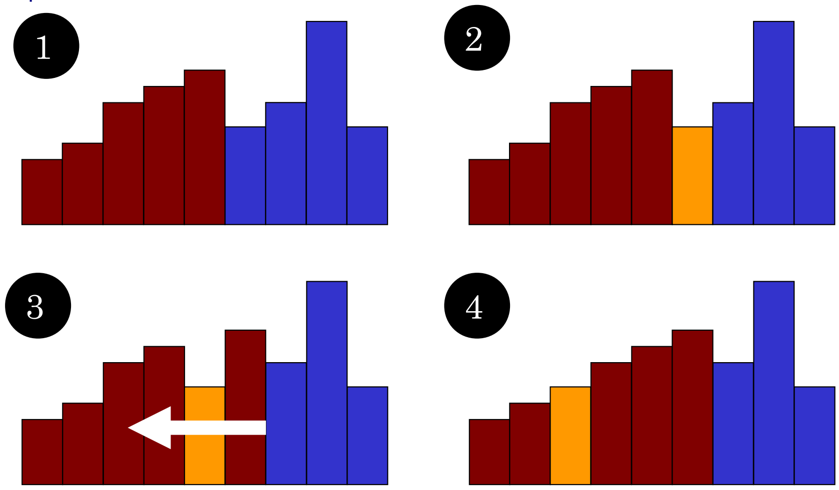
(Linear) insertion sort

- Algorithm is in-place!
- Split the array to be sorted $A[0,\dots,n-1]$ in 2 parts
  - $A[0,\dots,i-1]$ which is sorted
  - $A[i,\dots,n-1]$ not ordered yet

  Initially $i=1$, in which case $A[0,\dots,0]$ (trivially) is ordered

procedure InsertionSort($A[0,\dots,n-1]$)
  for $i=1$ **to** $n-1$ **do**
    Insert $A[i]$ in the right (=sorted) position in $A[0,\dots,i-1]$ 20.6

Example: Visualization of Insertion-sort



20.7

Worst case analysis of Insertion-sort

procedure InsertionSort($A[0,\dots,n-1]$)
  for $i=1$ **to** $n-1$ **do**
    $j \leftarrow i$; $x \leftarrow A[i]$
    while $j \geq 1$ **and** $A[j-1] > x$ **do**
      $A[j] \leftarrow A[j-1]$; $j \leftarrow j-1$
    $A[j] \leftarrow x$

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

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

- t_5: I pass
- t_6: $n-1$ pass
- Total: $t_2+t_3+t_4+t_5+t_6 = 3(n-1)+(n^2-n) = n^2+2n-3$ Thus $O(n^2)$ in the worst case… *but only if the sequence is almost sorted* 20.8

1.3 Selection sort

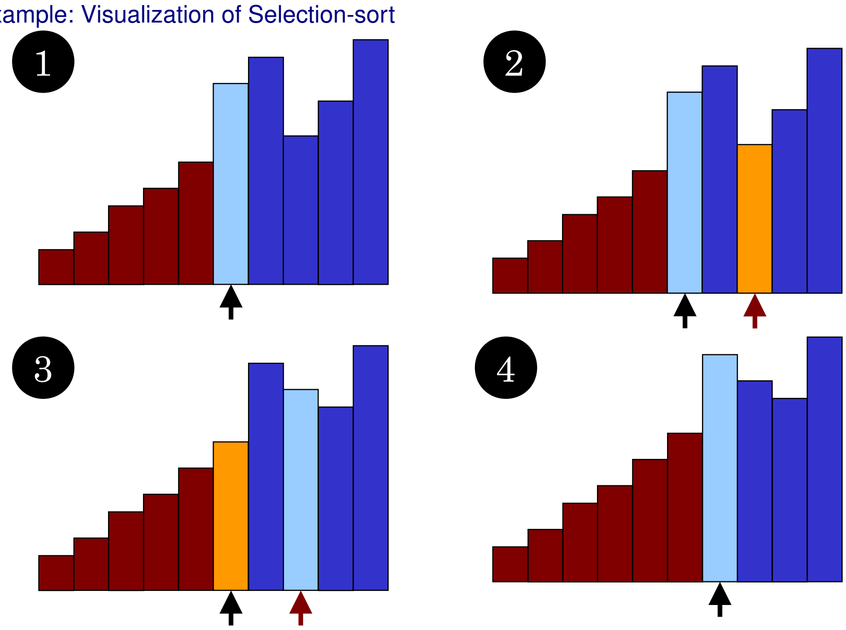
(Straight) selection sort

- Algorithm is in-place!
- Split the array to be sorted $A[0,\dots,n-1]$ in 2 parts
  - $A[0,\dots,i-1]$ which is sorted (all elements smaller than or equal to $A[i,\dots,n-1]$)
  - $A[i,\dots,n-1]$ not sorted yet

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

procedure SelectionSort($A[0,\dots,n-1]$)
  for $i=0$ **to** $n-2$ **do**
    Find the minimal element $A[j]$ in $A[i,\dots,n-1]$
    Swap locations of $A[i]$ and $A[j]$ 20.9

Example: Visualization of Selection-sort



20.10

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

20.11

1.4 Divide-and-conquer

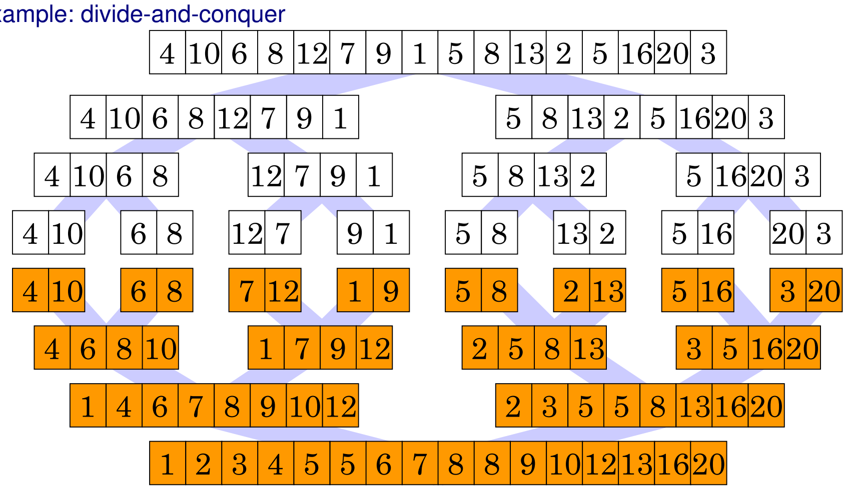
The principle of divide-and-conquer for algorithms construction

- divide: split up the problem into smaller, independent sub-problems
- conquer: solve sub-problems recursively (or directly if trivial)
- combine the solutions of sub-problems to solve the original problem

Sv. *söndra-och-härska*

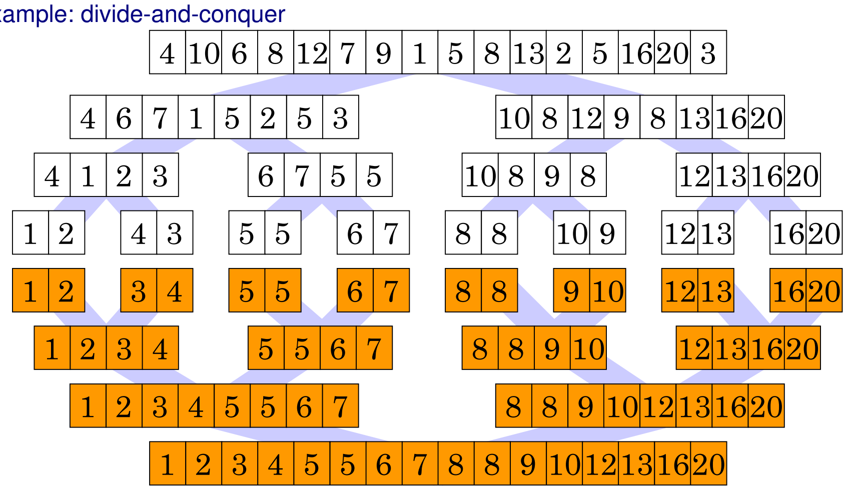
20.12

Example: divide-and-conquer



20.13

Example: divide-and-conquer



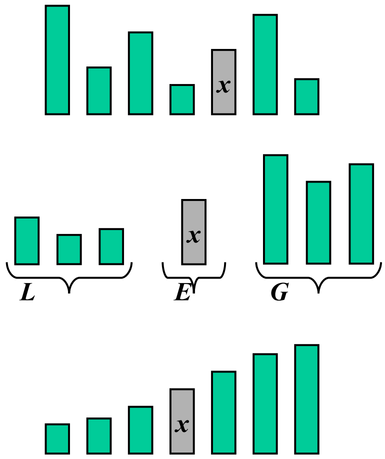
20.14

1.5 Quick-sort

Quick-sort

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

- divide: select randomly an element x (called pivot) and partition S to
  - L elements smaller than x
  - E elements equal to x
  - G elements greater than x
- conquer: sort L and G
- combine L, E and G



20.15

Partitioning

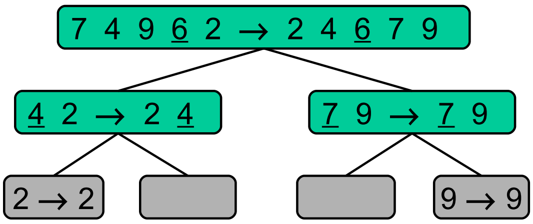
- We partition the input data sequence as follows:
  - We remove, i turn and order, each element y from S and
  - We insert y in L, E or G depending on the result of the comparison with pivot element x
- Each insertion and removal performed in the beginning or end of a sequence, and thus takes $O(1)$ time
- Thus, the partition step takes in quick-sort $O(n)$ 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
```

20.16

Quick-sort tree

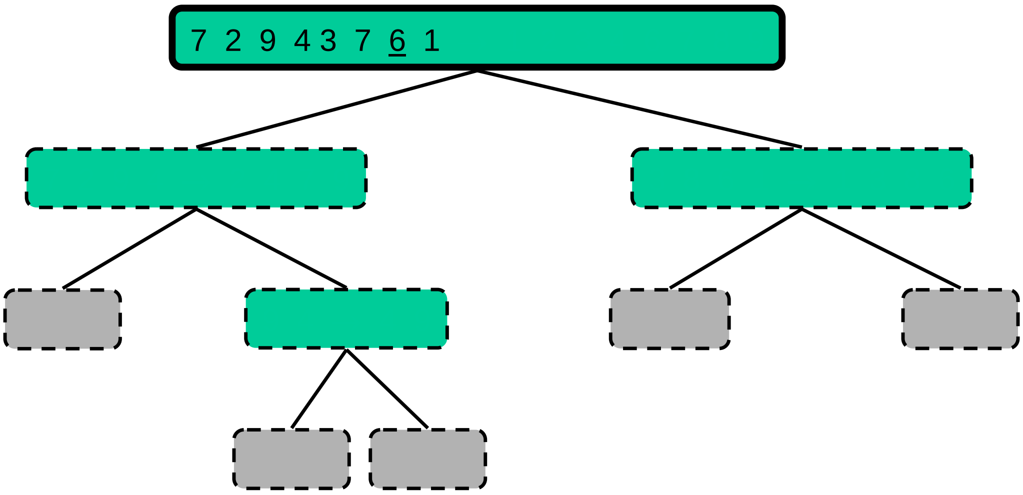
- The execution of quicksort can be visualized as a binary tree
  - Each node represents a recursive call to quicksort and stores
    - unsorted sequence before the execution and its pivot
    - Sorted sequence after the execution
  - The root is the originating call
  - The leaves are calls on partial sequences of size 0 or 1



20.17

Example: Execution of quick-sort

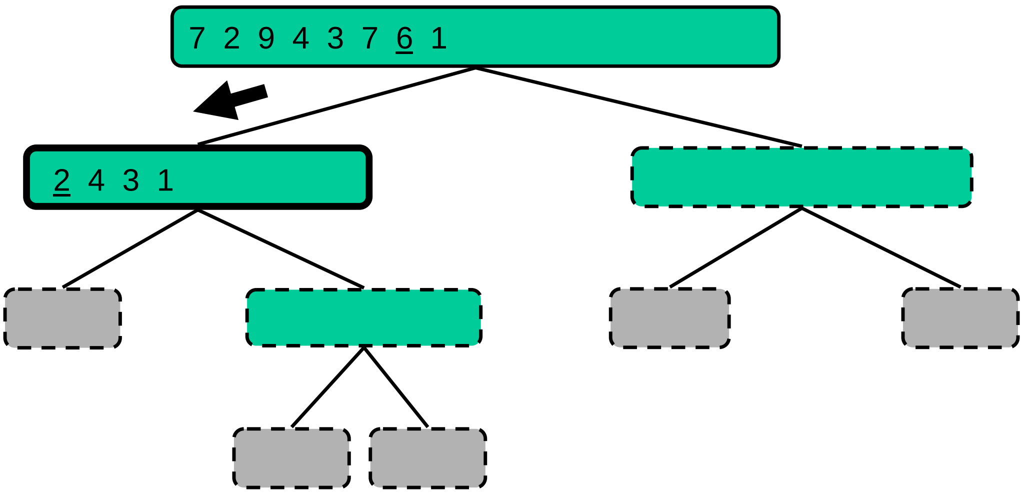
- Select a pivot



20.18

Example: Execution of quick-sort

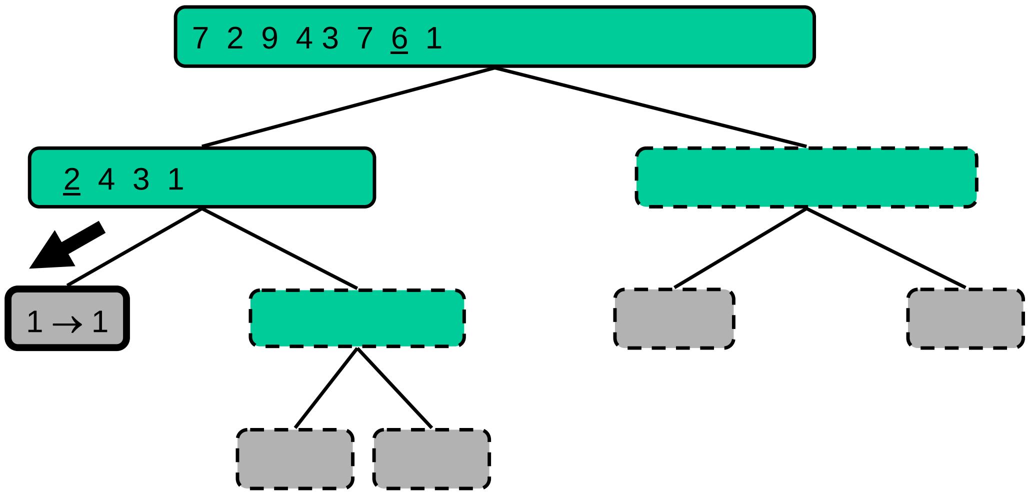
- Partitioning, recursive call, selection of pivot



20.19

Example: Execution of quick-sort

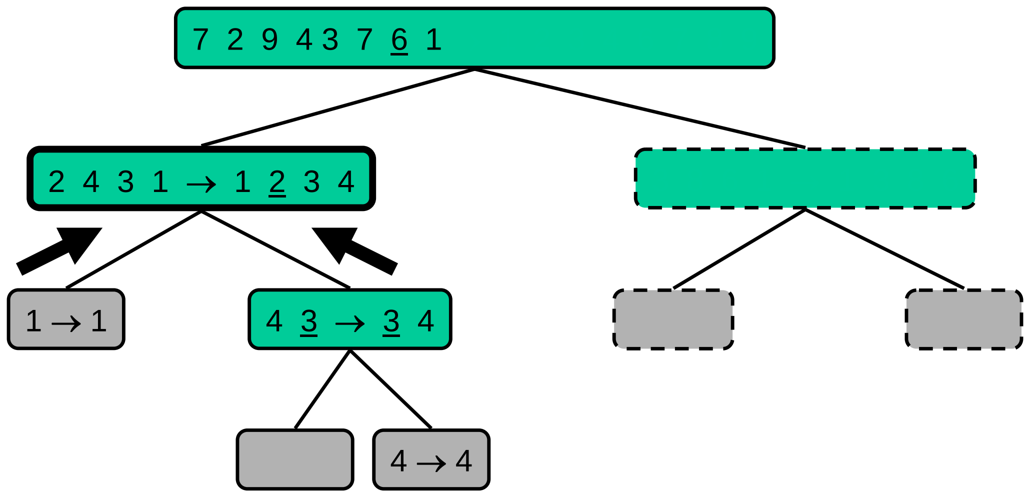
- Partitioning, recursive call, base case



20.20

Example: Execution of quick-sort

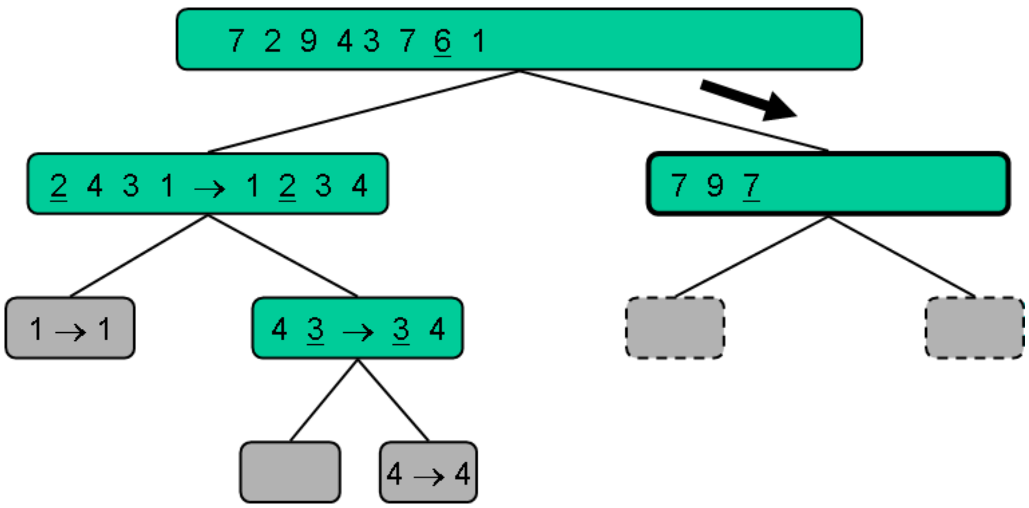
- Recursive call, . . . , base case, combination



20.21

Example: Execution of quick-sort

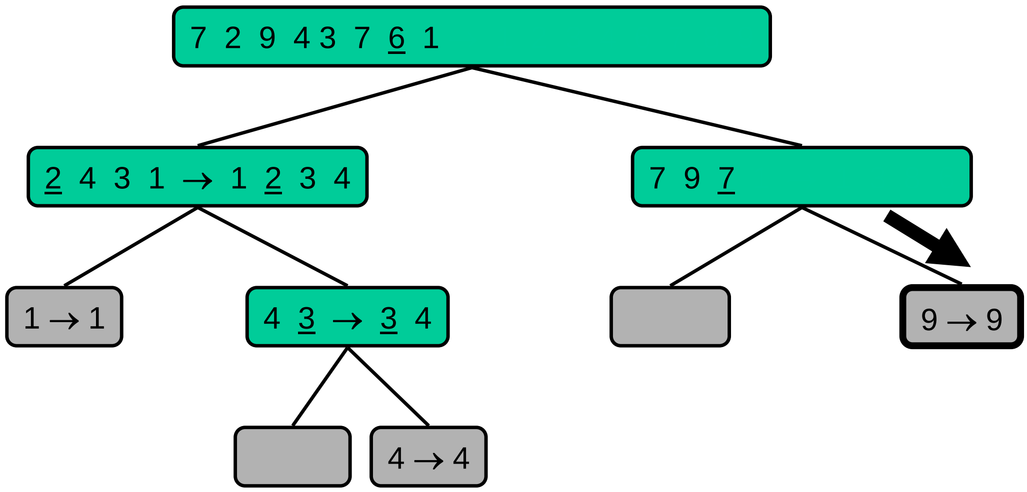
- Recursive call, choice of pivot



20.22

Example: Execution of quick-sort

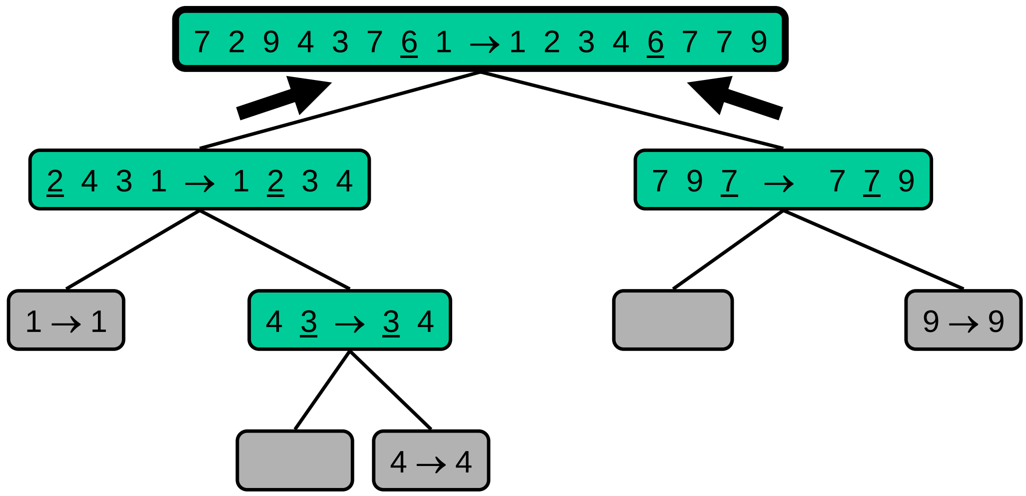
- Partitioning, . . . , recursive call, base case



20.23

Example: Execution of quick-sort

- combine, combine



20.24

Execution time in the worst case

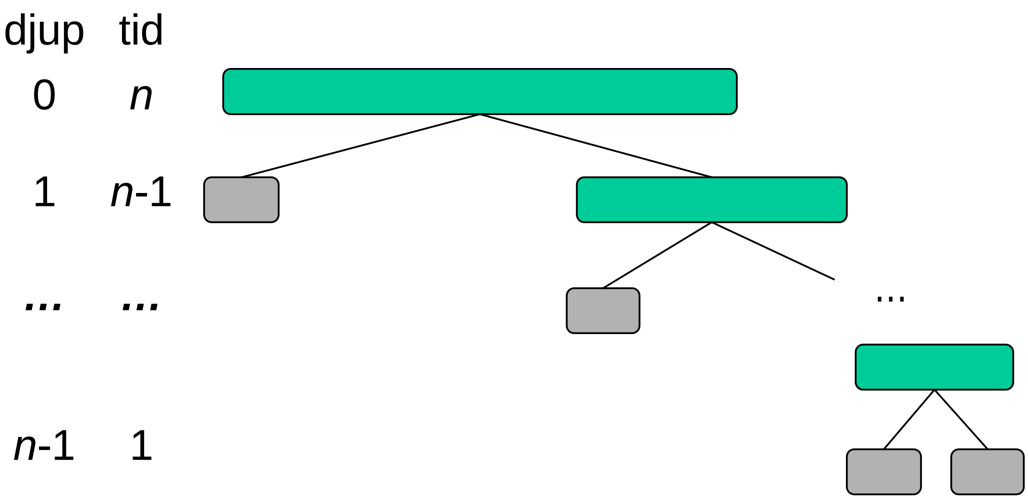
- The worst case for quick-sort occurs when the pivot element is a unique minimum or maximum element
- one of L or G has size $n-1$ and the other has size 0
- The execution time is proportional to the sum

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

- Thus, the worst case time for quick-sort is $O(n^2)$

20.25

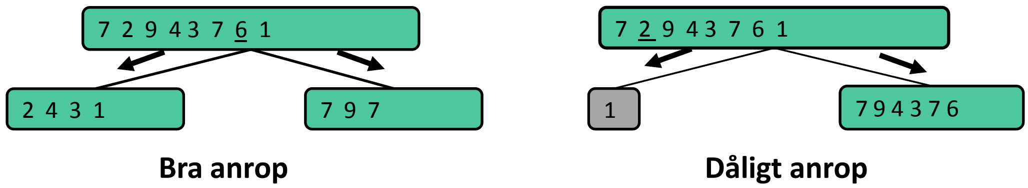
Execution time in the worst case



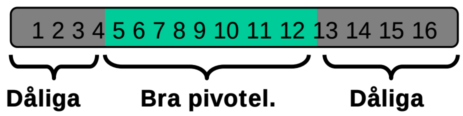
20.26

Expected execution time

- Consider a recursive call to quicksort on a sequence of size s
  - Good call: the sizes of L and G are both $< 3s/4$
  - Bad call: one of L and G has size $\geq 3s/4$



- A call is good with probability 1/2
  - Half of all possible pivot elements lead to a good call:



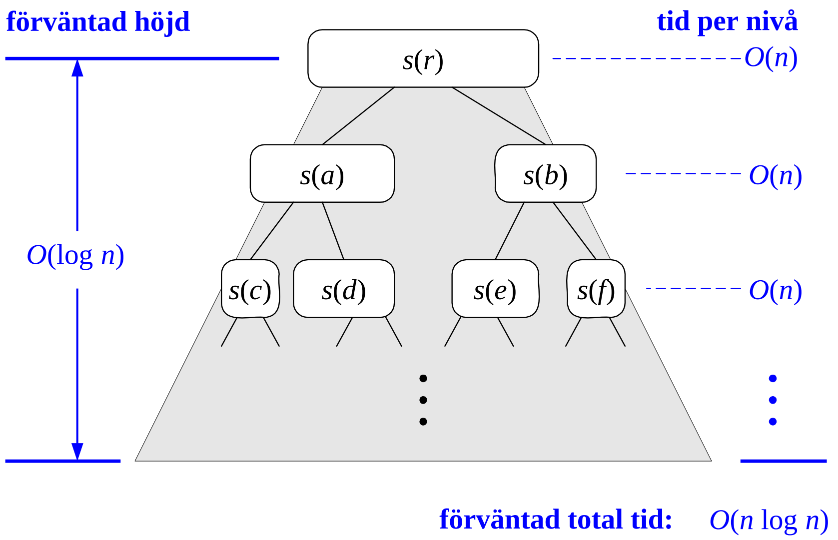
20.27

Expected execution time

- Probabilistic fact: the expected number of coin flips needed to get tails of k times is $2k$
- For a node at depth i, we expect
  - $i/2$ ancestors are good call
  - the size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Thus, we have
  - For a node at depth $2\log_{4/3} n$, then expected size of input data is 1
  - The expected height for quick-sort tree is $O(\log n)$
- The amount of work performed in the nodes at the same depth is $O(n)$
- Thus, the expected execution time for quick-sort is $O(n \log n)$

20.28

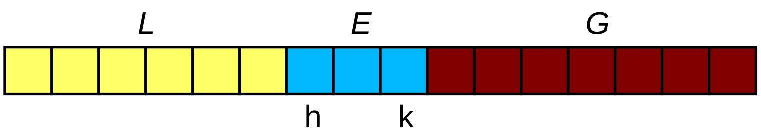
Expected execution time



20.29

Quick-sort with constant extra memory

- Quick-sort can be implemented to run *in-place*
- In the partitioning step, we use replacement operations to arrange the elements of the input sequence so that:
  - the elements smaller than the pivot element have rank less than h
  - the elements equal to the pivot element have rank between h and k
  - the elements greater than the pivot element have rank greater than k



20.30

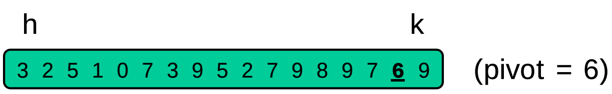
Algorithm for quick-sort with constant extra memory

procedure InPlaceQuickSort(S,l,r)
 if $l \geq r$ **then return**
 $i \leftarrow$ random integer between l and r
 $x \leftarrow S.$ElemAtRank(i)
 $(h,k) \leftarrow$InPlacePartition(x)
 InPlaceQuickSort$(S,l,h-1)$
 InPlaceQuickSort$(S,k+1,r)$

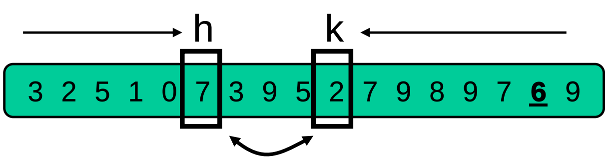
20.31

Partitioning with constant extra memory

- Perform partitioning using 2 indexes to split S into L and $E \cup G$ (A similar method can be used to split $E \cup G$ into E and G)



- Repeat the process until h and k meet/intersect:
  - Swipe h to the right until an element $\geq$ pivot element is found
  - Swipe k to the left until an element $<$ pivot element is found
  - swap the elements at locations h and k



20.32

2 Selection

2.1 Introduction

The selection problem

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

- We can sort the sequence in $O(n \log n)$ time, then index the ith element in constant time.

i=3 | 7 4 9 6 2 → 2 4 6 7 9

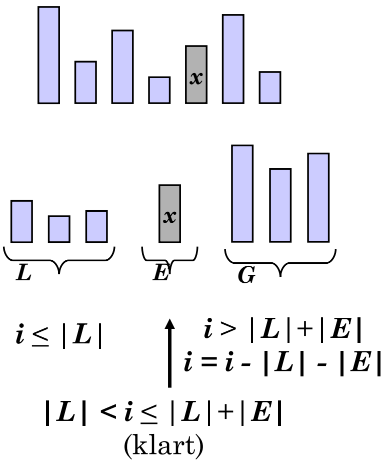
- Can we solve the selection problem faster? 20.33

2.2 Quick-select

Quick-select

Quick-Select is a randomized selction problem based on the paradigm *prune-and-search*:

- *Prune*: Select x randomly and partition S to
  - L elements lower than x
  - E elements equal to x
  - G elements greater than x
- *Search*: depending on i, either the solution exists in E or we need to continue recursively in one of L or G.



20.34

Partitioning

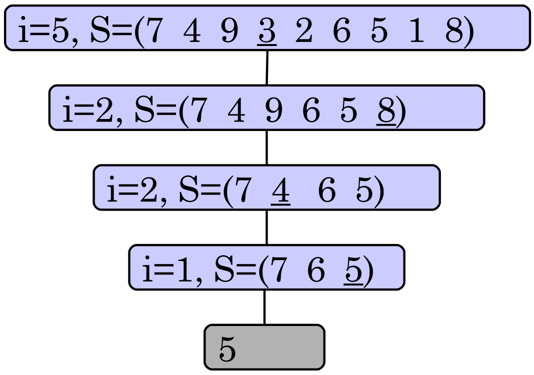
- We partition the input sequence as in the algorithm for quick-sort:
  - We remove, i turn and order, each element y from S and
  - We insert y in L, E or G depending on the result of comparison with the pivot element x
- Insertion and deletion are performed at the beginning or end of a sequence, thus each takes $O(1)$ time
- Thus, 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
```

20.35

Visualization of Quick-select

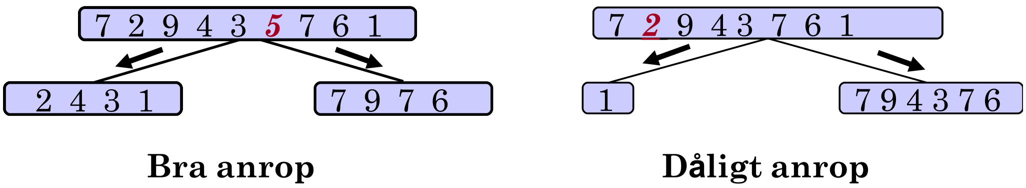
- The execution of quick-select can be visualized with the help of a recursion path
  - Each node represents a recursive call to quick-select and stores i and the remaining sequence S



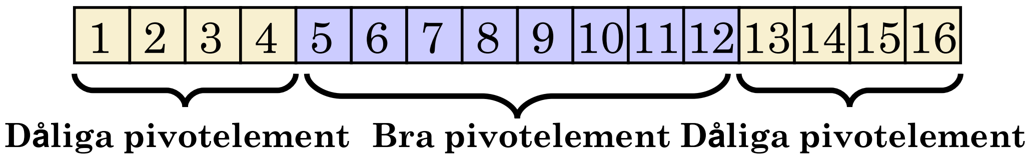
20.36

Expected execution time

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



- A call is good with probability 0.5
  - Half of all possible pivot elements lead to good calls:



20.37

Expected execution time

- Probabilistic fact: The expected number of coin flips needed to get tail once is two
- Probabilistic fact: The expected value is a linear function:
  - $E(X+Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$ for each constant c
- Let $T(n)$ be the expected execution time for quick-select
- By the second fact, we get $T(n) \leq b \cdot n \cdot g(n) + T(3n/4)$ where
  - b is a constant
  - $g(n)$ is the expected number of calls before a good call occurs

20.38

Expected execution time

- Thus
  - $T(n) \leq b \cdot n \cdot g(n) + T(3n/4)$
- Through the first fact, we get
  - $T(n) \leq 2 \cdot b \cdot n + T(3n/4)$
- $T(n)$ is a geometric serie:
  - $T(n) \leq 2 \cdot b \cdot n + 2 \cdot b \cdot n \cdot (3/4) + 2 \cdot b \cdot n \cdot (3/4)^2 + 2 \cdot b \cdot n \cdot (3/4)^3 + \ldots$
- Thus, $T(n) \in O(n)$
- We can solve the selection problem in expected time $O(n)$ (the worst case time is $O(n^2)$)

20.39