Lecture 21

Heap-sort, merge-sort. Lower limits for sorting. Sorting in linear time?

TDDD86: DALP

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

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

21.1

Content

Innehåll

21.2

1 Sorting

1.1 Heap-sort

Sorting with priority queues

- Use a priority queue to sort a collection of comparable elements
  - Insert an element with a serie of insertion operations
  - Remove the elements in sorted order with a series of operations removeMin
- The runtime depends on the implementation of the priority queue:
  - Unsorted sequences give a selection sort and $O(n^2)$ time
  - Sorted sequences give insertion sorting and $O(n^2)$ time

```
procedure PQSORT(S)
    P ← empty priority queue
    while ¬S.ISEMPTY() do
        e ← S.REMOVE(S.FIRST())
        P.INSERT(e)
    while ¬P.ISEMPTY() do
        e ← P.REMOVEMIN()
        S.INSERTLAST(e)
```

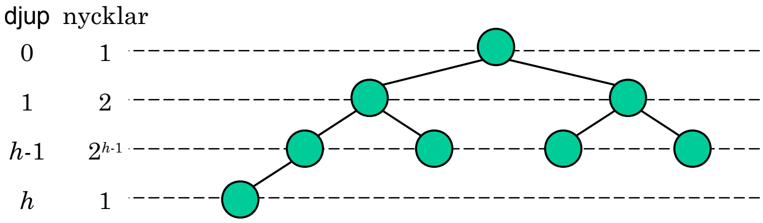
The height of a heap

Proposition 1. The height of a heap storing n keys is $O(\log n)$

Bevis. We use a heap as a complete binary tree.

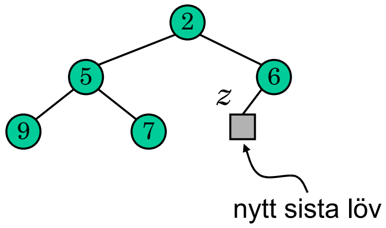
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots h-1$ and at least one key in depth h we get $n \geq 1+2+4+\ldots+2^{h-1}+1$
- Thus $n \geq 2^h$, i.e. $h \leq log_2 n$

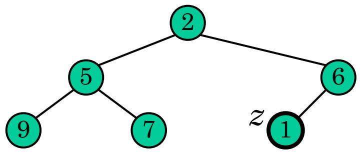
□



Insertion in a heap

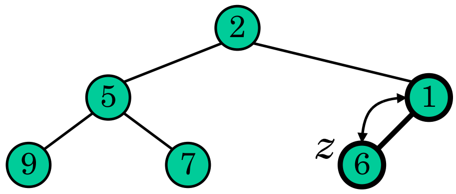
- The method insert in an ADT priority queue corresponds to the insertion of key k in the heap
- The insertion algorithm consists of three steps
  - Find the place to insert z (the new last leaf)
  - Store k in z
  - Reset the heap property

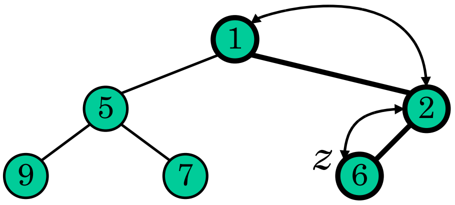




Upheap

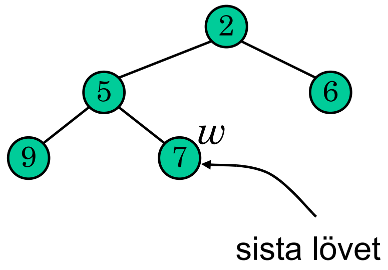
- After inserting a new key k, it is not certain that the heap property is still fulfilled
- The method upheap restores the heap property by swapping k along the upward path from the inserted node
- upheap terminates when the key k reaches the root or a node whose parent has a key that is not greater than k
- Since the height of a heap is $O(\log n)$, upheap runs in $O(\log n)$ time

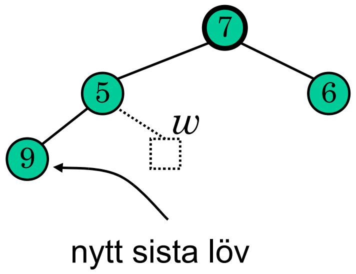




Removal from a heap

- Method removeMin consists in removing the root key from the heap
- Removal algorithm consists of three steps
  - Replace the root key with the key in the last leaf w
  - remove w
  - Reset the heap property





Downheap

- After replacement of the root key with key k from the last leaf, it is not certain that the heap property is still fulfilled
- Method downheap restores the heap property by swapping k along the downward path of the insertion node
- downheap terminates when the key k reaches a leaf or a node whose children have keys that are not less than k
- Since the height of a heap is $O(\log n)$, downheap runs in time $O(\log n)$

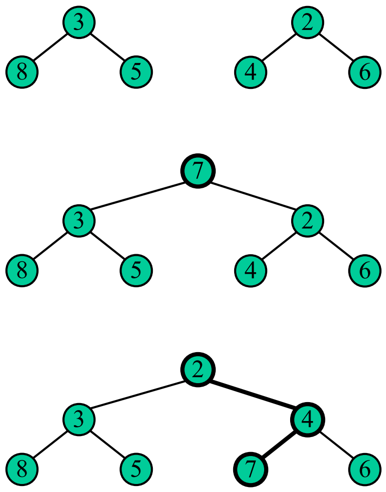
Heap-sort

- Consider a priority queue with n elements implemented in terms of a heap
  - memory utilization is $O(n)$
  - insert and removeMin run in $O(\log n)$ time
  - size, isEmpty and min run in $O(1)$ time

- Upon the utilization of heap-based priority queue we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms

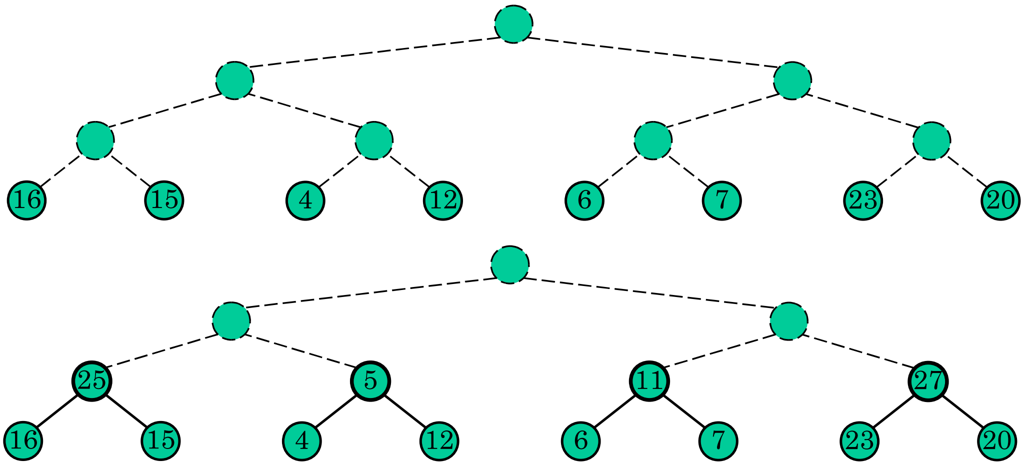
Combine 2 heaps

- Given 2 heaps and a key k
- Create a new heap where the root node stores the key k, and the two given heaps as subtrees
- Run downheap to reset the heap property

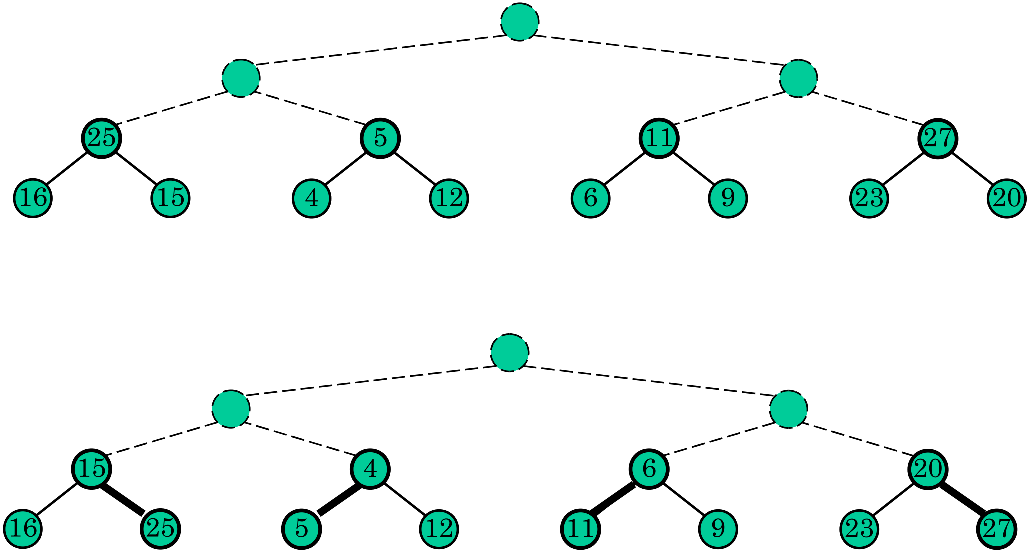


Example: construction of a heap bottom-upp

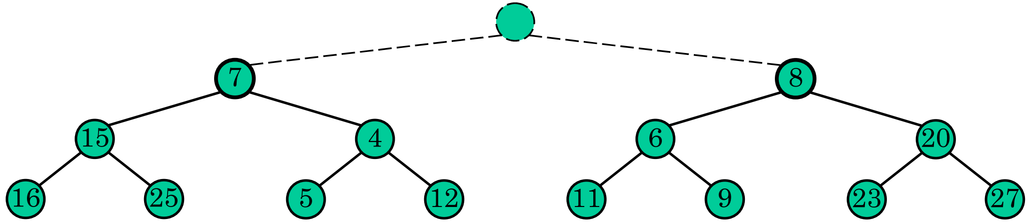
10	7	8	25	5	11	27	16	15	4	12	6	7	23	20

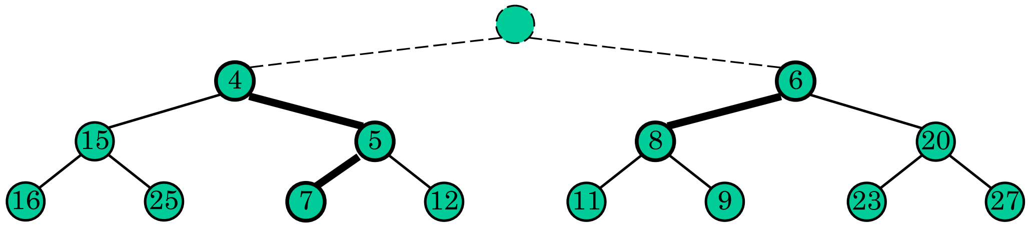


Example: construction of a heap bottom-upp

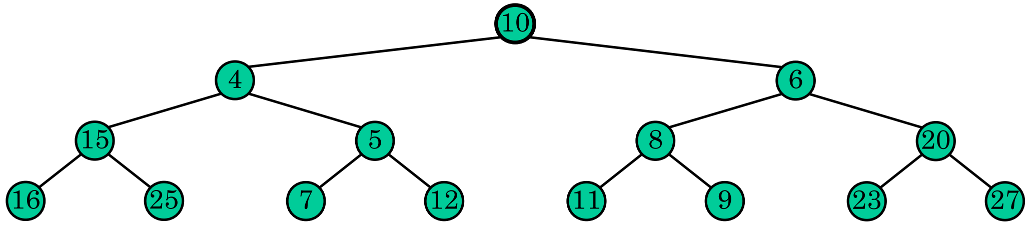


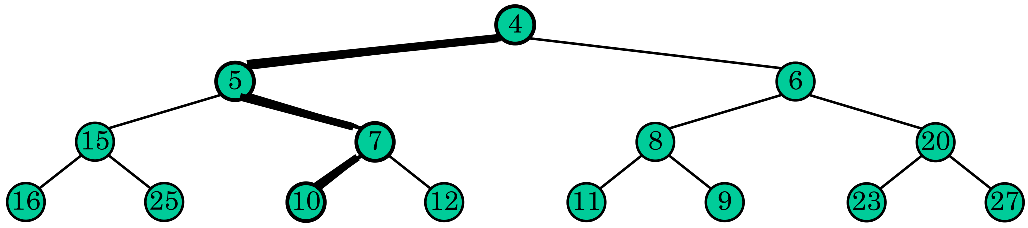
Example: construction of a heap bottom-upp





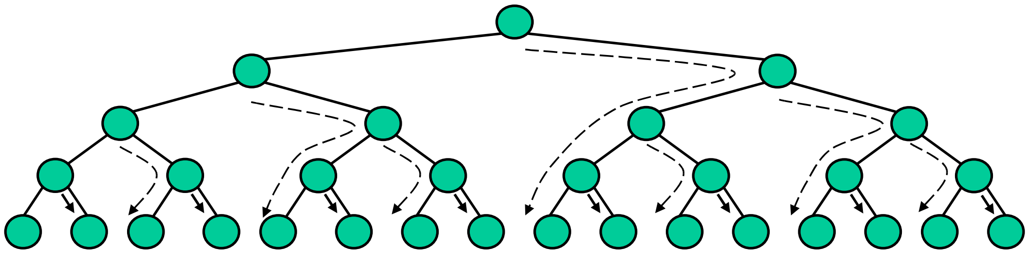
Example: construction of a heap bottom-upp





Analysis

- We visualize the worst case time for a call to downheap with a path that first goes to the right, and then repeatedly go left to the bottom of the heap
- Since each node is traversed by at most two such paths, the total number of paths is $O(n)$
- Thus, the time to construct a heap bottom-upp is $O(n)$
- This construction method is faster than the n repeated deposits and makes the first phase of heap-sort more efficient



1.2 Merge-sort

Divide and conquer

- Merge-sort is a sort algorithm based on divide and conquer
- Like the heap-sort

  - the execution time is $O(n \log n)$
- different heap-sort
  - does not use priority queues to help
  - access the data in a sequential manner (suitable to sort the data on disk)

Merge-sort

Merge-sort on an input sequence S having n elements is performed in 3 steps:

- Divide: split S into 2 sequences S_1 and S_2 each with $n/2$ elements
- Conquer: sort S_1 and S_2 recursively
- Combine: merge S_1 and S_2 in a unique sorted sequence

```
procedure MERGESORT($S$)
    if $S$.SIZE() $> 1$ then
        $(S_1, S_2) \leftarrow$PARTITION($S$.SIZE()$/2$)
        MERGESORT($S_1$)
        MERGESORT($S_2$)
        $S \leftarrow$MERGE($S_1, S_2$)
```

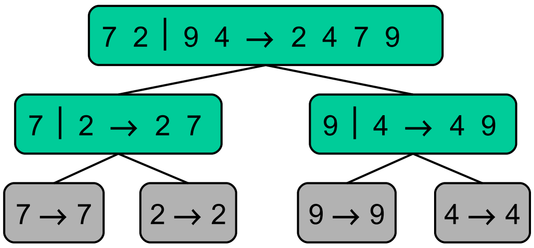
Merge two sorted sequences

- Merge 2 sequences A and B to form a seuqnce S containing the union of elements in A and B
- Merging 2 sorted sequences, each with $n/2$ elements, implemented with double linked lists takes $O(n)$ time

```
function MERGE($A, B$)
    $S \leftarrow$ empty sequence
    while $\neg A$.ISEMPTY() $\wedge \neg B$.ISEMPTY() do
        if $A$.FIRST.ELEMENT() $<$ $B$.FIRST.ELEMENT() then
            $S$.INSERTLAST($A$.REMOVE($A$.FIRST()))
        else
            $S$.INSERTLAST($B$.REMOVE($B$.FIRST()))
    while $\neg A$.ISEMPTY() do
        $S$.INSERTLAST($A$.REMOVE($A$.FIRST()))
    while $\neg B$.ISEMPTY() do
        $S$.INSERTLAST($B$.REMOVE($B$.FIRST()))
    return $S$
```

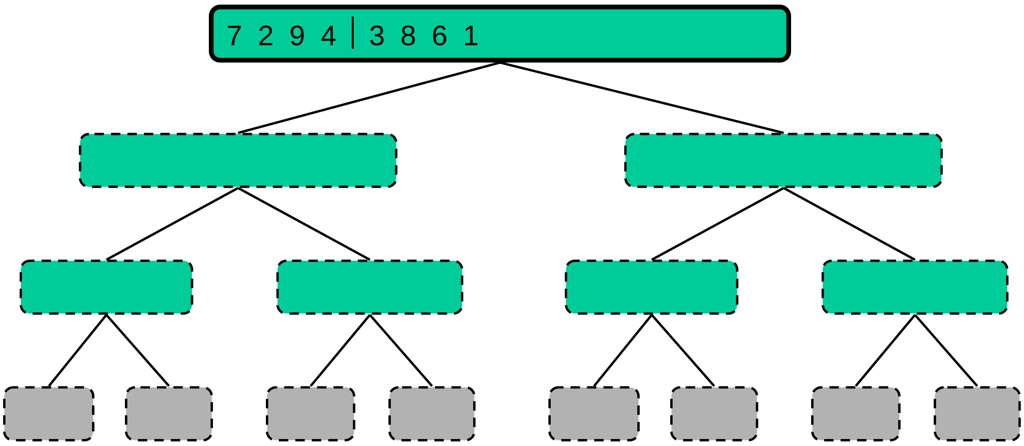
Merge-sort tree

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



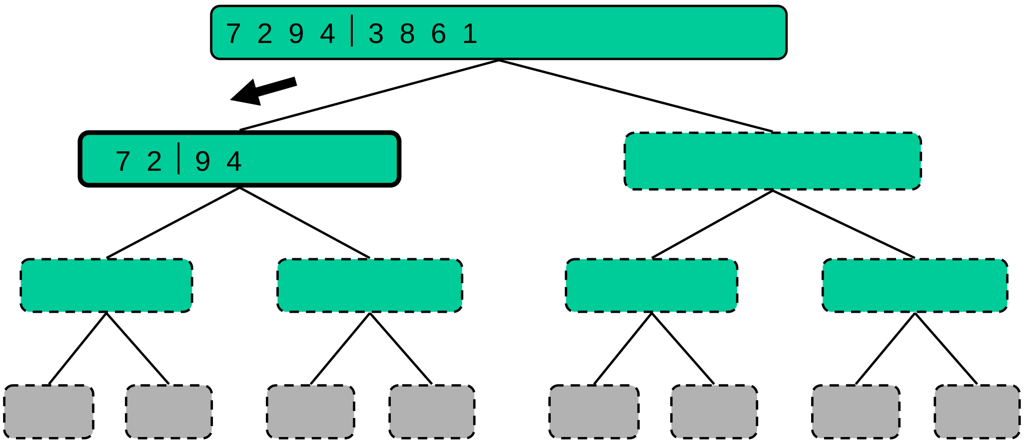
Example: Execution of merge-sort

- Partitioning



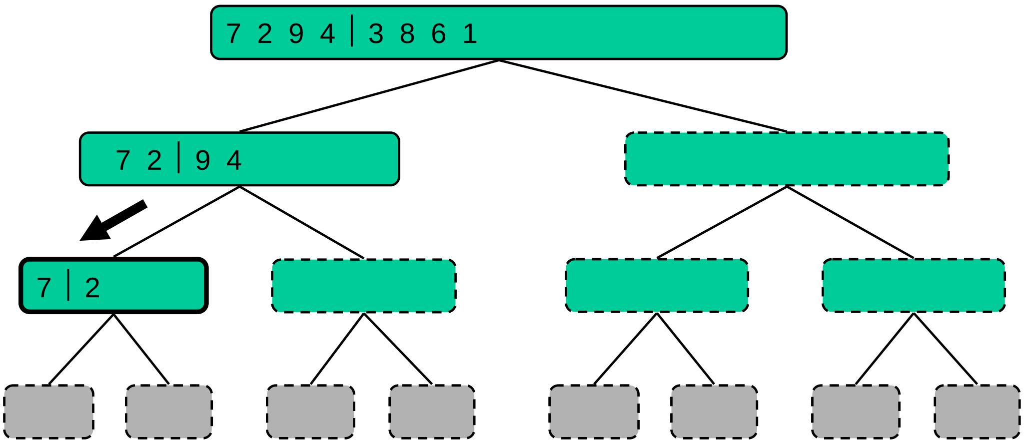
Example: Execution of merge-sort

- Recursive call, partitioning



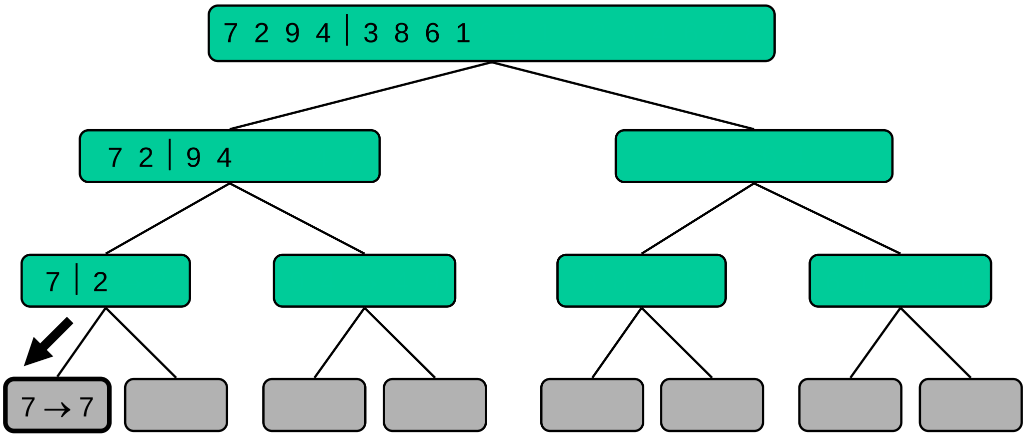
Example: Execution of merge-sort

- Recursive call, partitioning



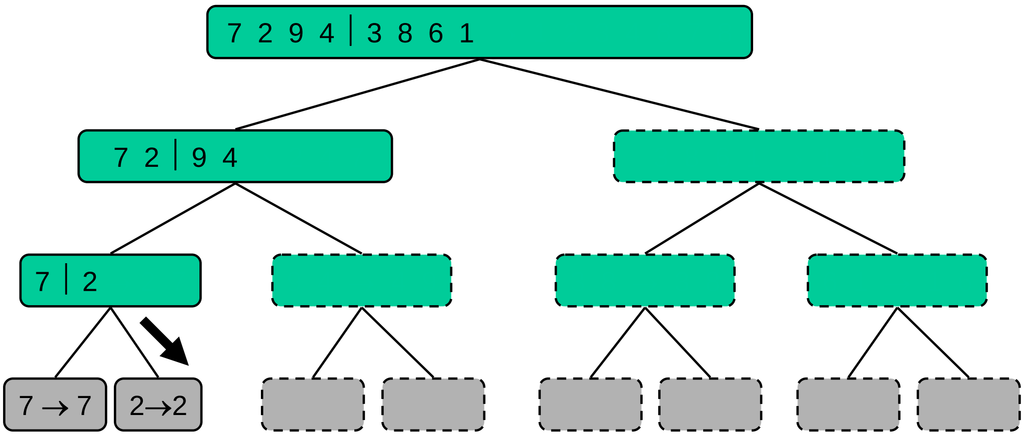
Example: Execution of merge-sort

- Recursive call, base case



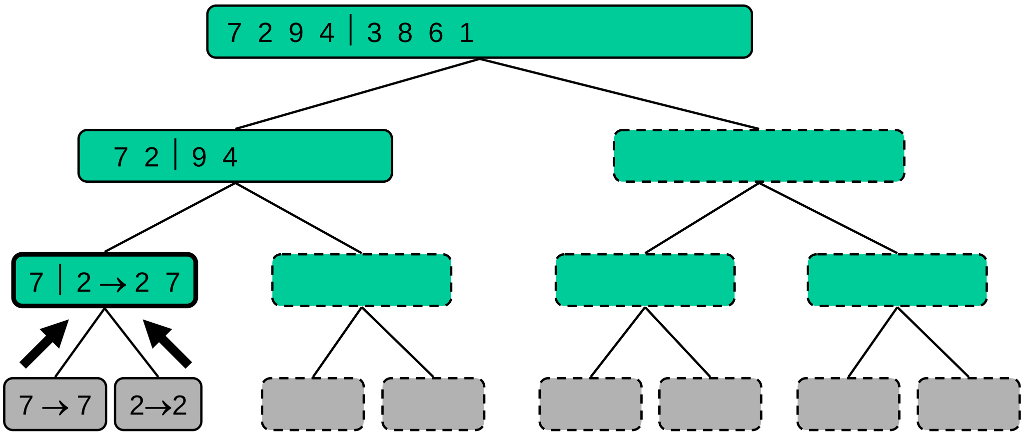
Example: Execution of merge-sort

- Recursive call, base case



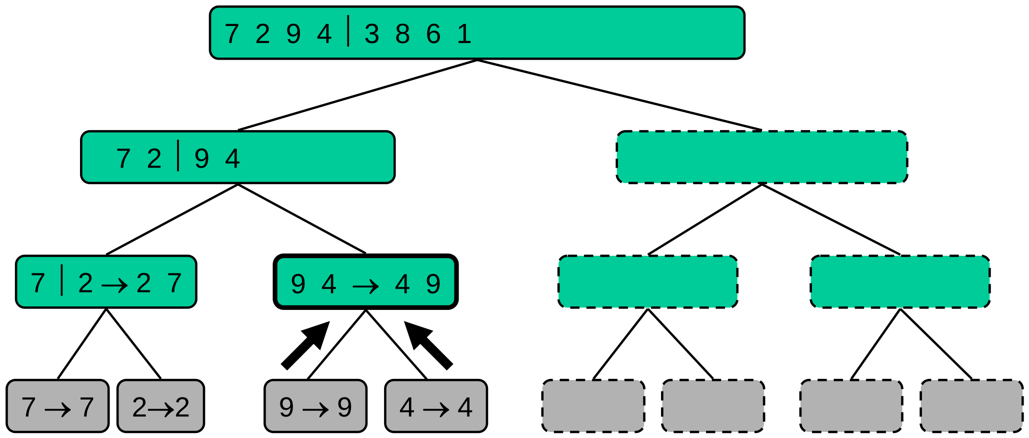
Example: Execution of merge-sort

- Merging



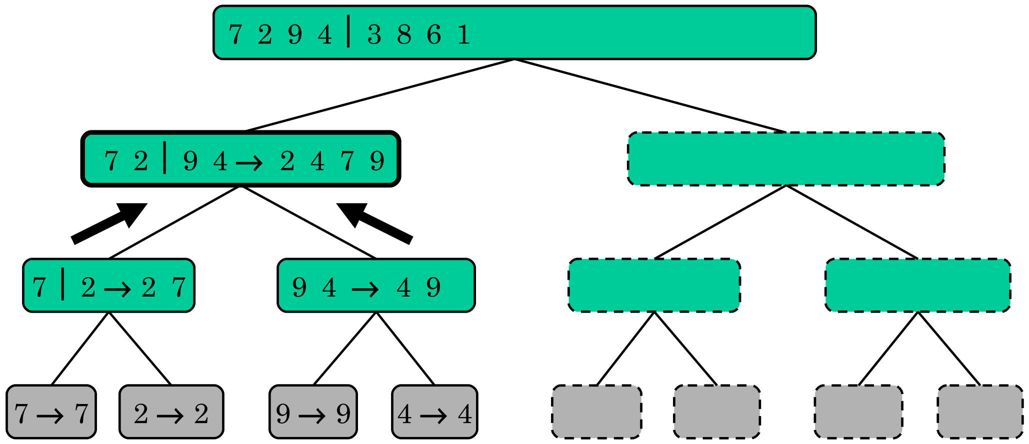
Example: Execution of merge-sort

- Recursive call, . . . , base case



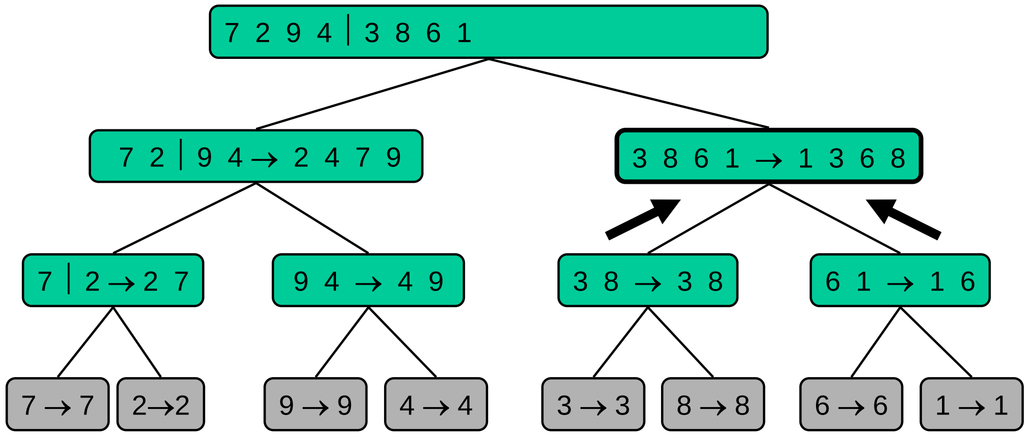
Example: Execution of merge-sort

- merging



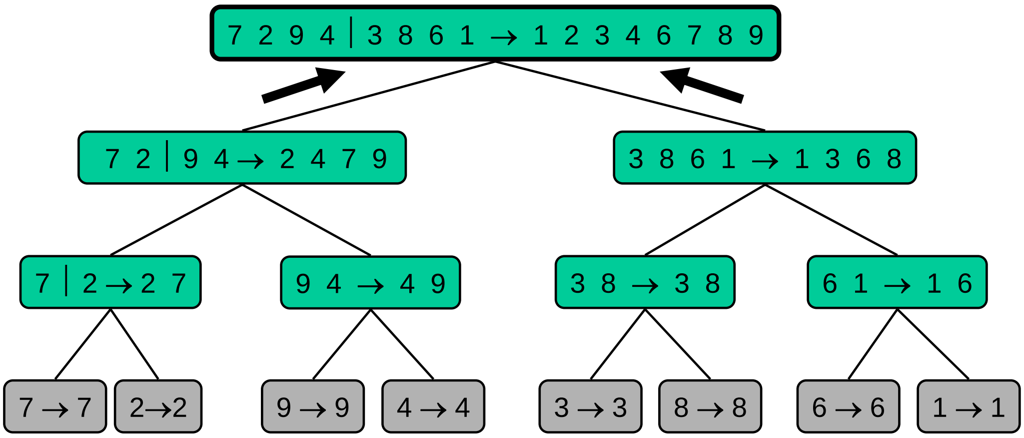
Example: Execution of merge-sort

- Recursive call, . . . , merging, merging



Example: Execution of merge-sort

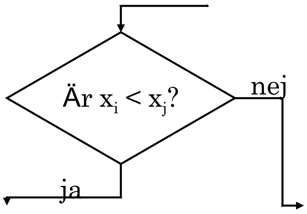
- merging



2 A lower limit for the comparison based sorting

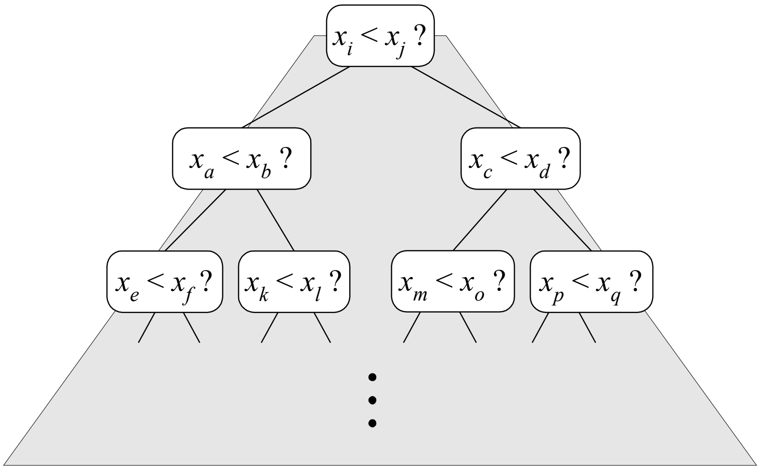
comparison based sorting

- Many sorting algorithms are *comparison-based*
  - They sort through comparisons between pairs of objects
  - Example: insertion-sort, selection-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore try to derive a lower limit for the execution time in the worst case for each algorithm using the comparison to sort n elements $x_1, x_2, \ldots, x_n$



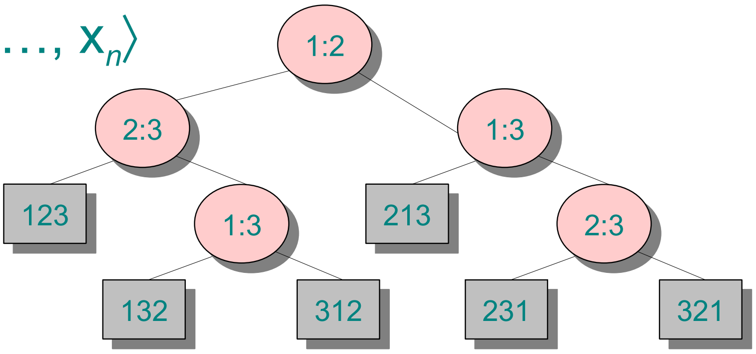
Calculating comparisons

- Let's just count comparisons
- Every possible execution of an algorithm is represented by a root-to-leaf path in a *decision tree*



Example: Decision tree

Sort $\langle x_1, x_2, \ldots, x_n \rangle$

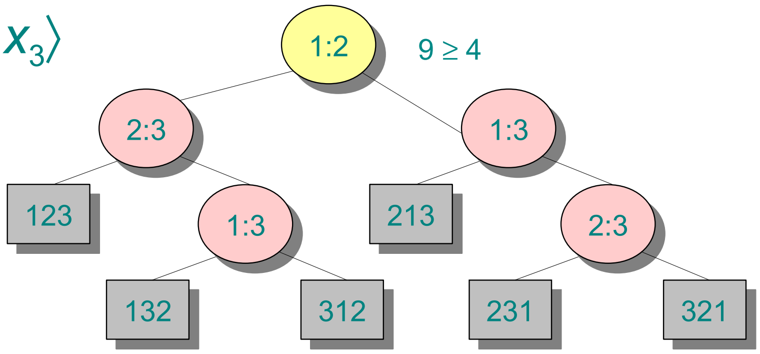


Each internal node is marked $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $x_i \leq x_j$
- The right subtree shows subsequent comparisons if $x_i \geq x_j$

Example: Decision tree

Sort $\langle x_1, x_2, x_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

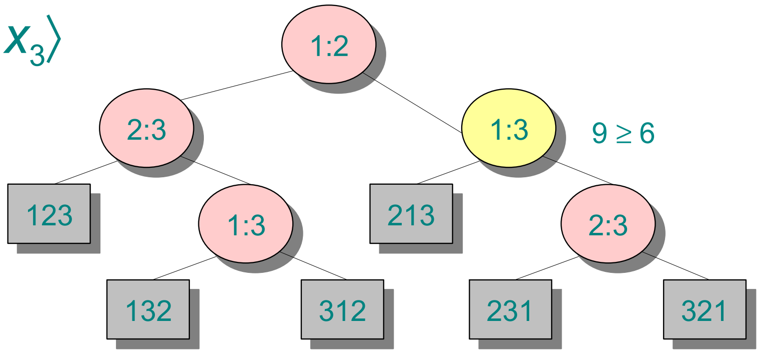


Each internal node is marked $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $x_i \leq x_j$
- The right subtree shows subsequent comparisons if $x_i \geq x_j$

Example: Decision tree

Sort $\langle x_1, x_2, x_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

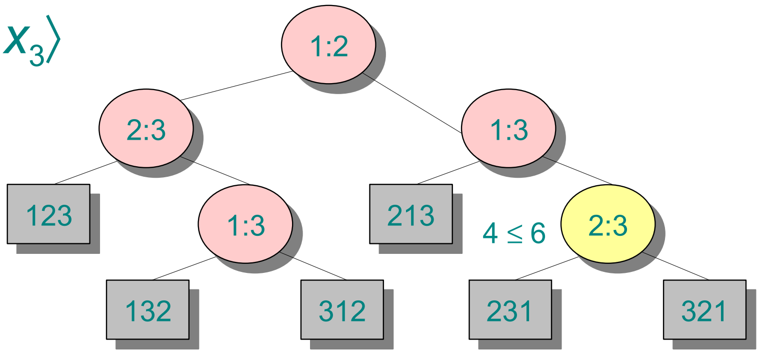


Each internal node is marked $i : j$ for $i, j \in \{1, 2, \ldots, n\}$

- The left subtree shows subsequent comparisons if $x_i \leq x_j$
- The right subtree shows subsequent comparisons if $x_i \geq x_j$

Example: Decision tree

Sort $\langle x_1, x_2, x_3 \rangle$
$= \langle 9, 4, 6 \rangle$:



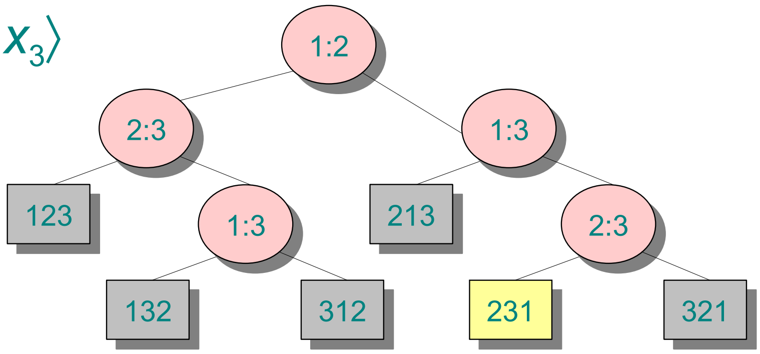
Each internal node is marked $i : j$ for $i, j \in \{1, 2, \dots, n\}$

- The left subtree shows subsequent comparisons if $x_i \leq x_j$
- The right subtree shows subsequent comparisons if $x_i \geq x_j$ 21.35

Example: Decision tree

Sort $\langle x_1, x_2, x_3 \rangle$

$= \langle 9, 4, 6 \rangle$:



$4 \leq 6 \leq 9$

Each leaf contains a permutation $\langle \pi(i), \pi(2), \dots, \pi(n) \rangle$ to indicate that the order $x_{\pi(1)} \leq x_{\pi(2)} \leq \dots \leq x_{\pi(n)}$ has been established 21.36

Decision tree model

A decision tree can model the execution of the comparison-based sorting algorithms:

- A tree for each size of the input data
- Consider the algorithm execution to be shared whenever two elements are compared
- The tree contains all comparisons along all the possible consequences of instructions
- The running time of the algorithm = the length of the path traversed
- The running time in the worst case = the height of the tree 21.37

The height of a decision tree

- The height of the decision tree is a lower limit on the execution time in the worst case
- Every possible permutation of the input should lead to a separate output leaf
- Since there is $n! = 1 \cdot 2 \cdot \ldots \cdot n$ leaves, the height of a tree is at least $\log(n!)$ 21.38

3 Sorting in linear time?

3.1 Counting-sort

Counting sort

```
Require: $A[1,\dots,n]$, where $A[j] \in \{1,2,\dots,k\}$
  function CountingSort($A$)
      Array to count : $C[1,\dots,k]$
      Array to store the result: $Res[1,\dots,n]$
      for $i \leftarrow 1$ to $k$ do
          $C[i] \leftarrow 0$
      for $j \leftarrow 1$ to $n$ do
          $C[A[j]] \leftarrow C[A[j]] + 1$                ▷ $C[i] = |\{nyckel = i\}|$
      for $i \leftarrow 2$ to $k$ do
          $C[i] \leftarrow C[i] + C[i-1]$                 ▷ $C[i] = |\{nyckel \leq i\}|$
      for $j \leftarrow n$ downto $i$ do
          $Res[C[A[j]]] \leftarrow A[j]$
          $C[A[j]] \leftarrow C[A[j]] - 1$
      return $Res$
```
21.39

Counting-sort

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4

Res:

Loop 1

A:

1	2	3	4	5
4	1	3	4	3

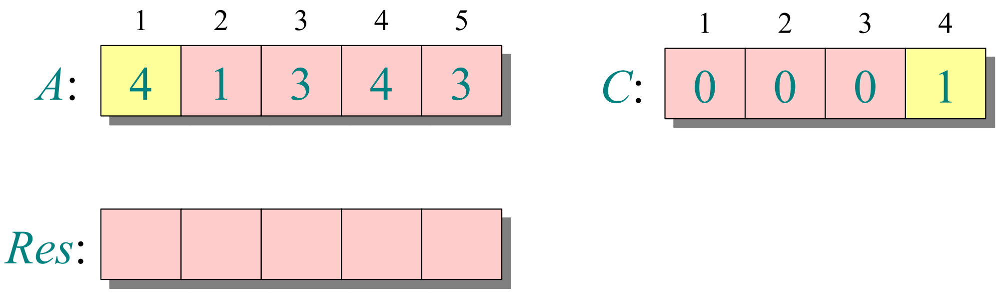
C:

1	2	3	4
0	0	0	0

Res:


```
for $i \leftarrow 1$ to $k$ do
  $C[i] \leftarrow 0$
```

Loop 2



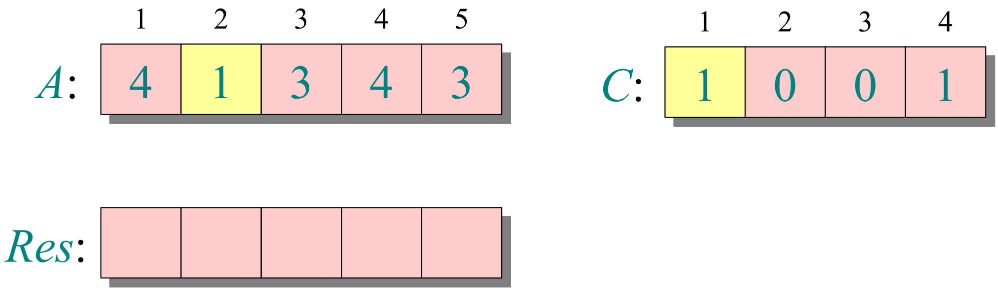
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

Res:					

```
for $j \leftarrow 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright\ C[i] = |\{\text{nyckel} = i\}|$
```

Loop 2



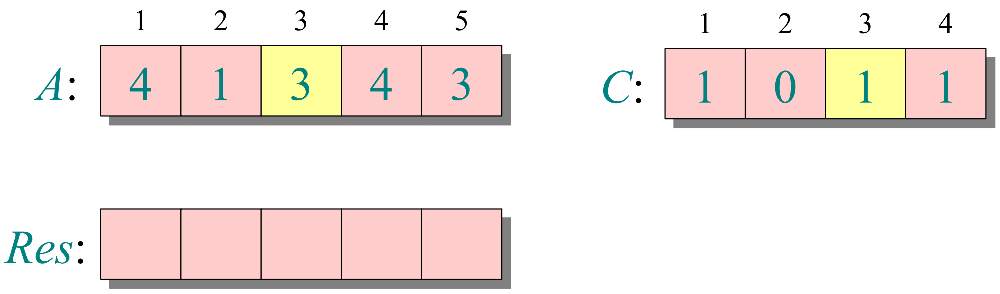
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	0	1

Res:					

```
for $j \leftarrow 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$   $\triangleright\ C[i] = |\{\text{nyckel} = i\}|$
```

Loop 2



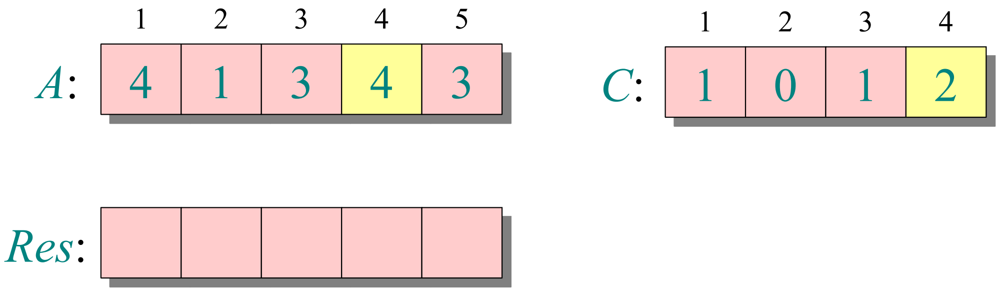
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

Res:					

```
for $j \leftarrow 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$  $\rhd$ $C[i] = |\{\text{nyckel} = i\}|$
```

Loop 2



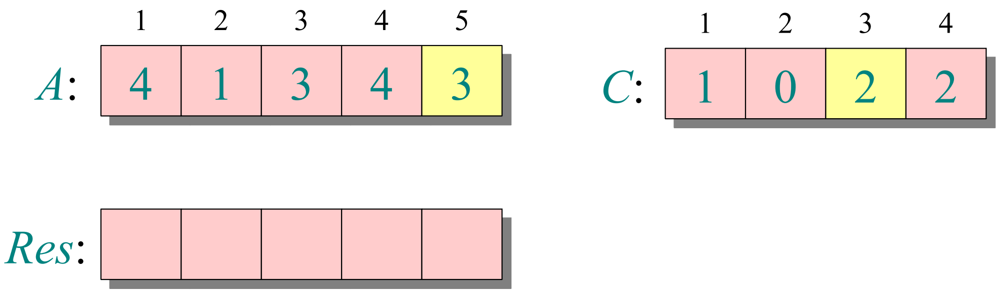
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	2

Res:					

```
for $j \leftarrow 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$  $\rhd$ $C[i] = |\{\text{nyckel} = i\}|$
```

Loop 2



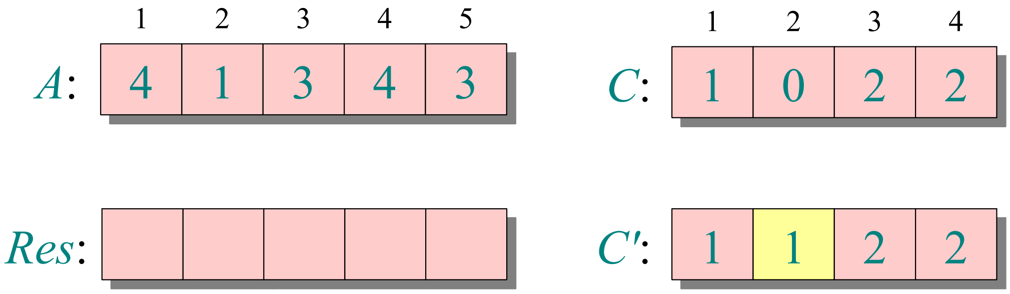
	1	2	3	4	5
A:	4	1	3	4	3
Res:					

	1	2	3	4
C:	1	0	2	2

```
for $j \leftarrow 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$   ▹ $C[i] = |\{\text{nyckel} = i\}|$
```

Example

Loop 3



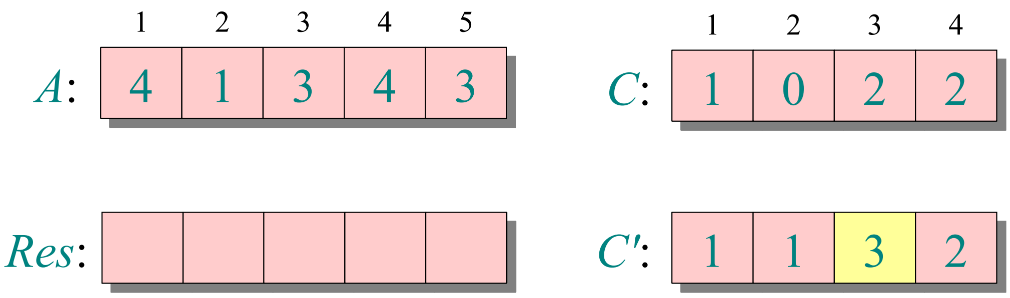
	1	2	3	4	5
A:	4	1	3	4	3
Res:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	2	2

```
for $i \leftarrow 2$ to $k$ do
    $C[i] \leftarrow C[i] + C[i-1]$   ▹ $C[i] = |\{\text{nyckel} \leq i\}|$
```

Example

Loop 3



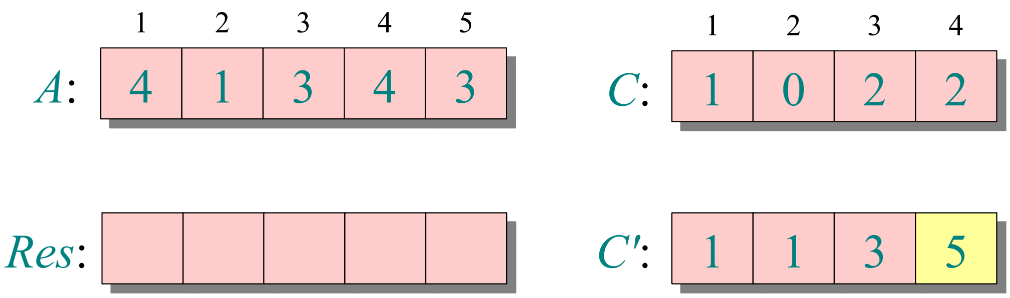
	1	2	3	4	5
A:	4	1	3	4	3
Res:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

```
for $i \leftarrow 2$ to $k$ do
    $C[i] \leftarrow C[i] + C[i–1]$    ▹ $C[i] = |\{\text{nyckel} \leq i\}|$
```

Example

Loop 3



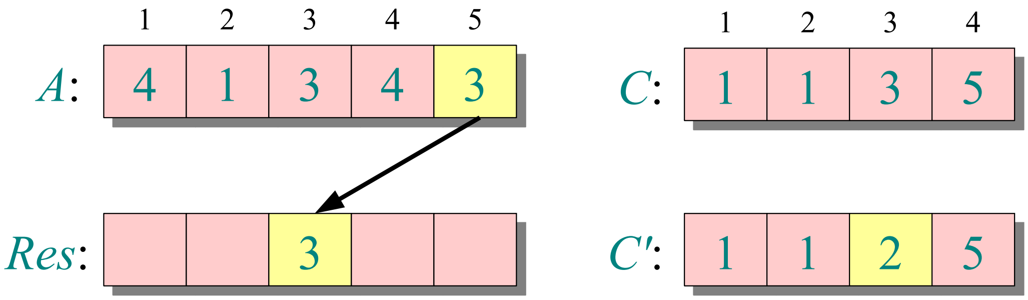
	1	2	3	4	5
A:	4	1	3	4	3
Res:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

```
for $i \leftarrow 2$ to $k$ do
    $C[i] \leftarrow C[i] + C[i–1]$    ▹ $C[i] = |\{\text{nyckel} \leq i\}|$
```

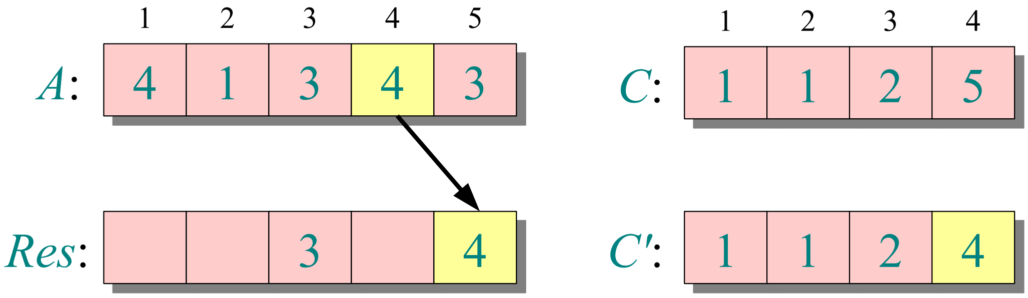
Example

Loop 4



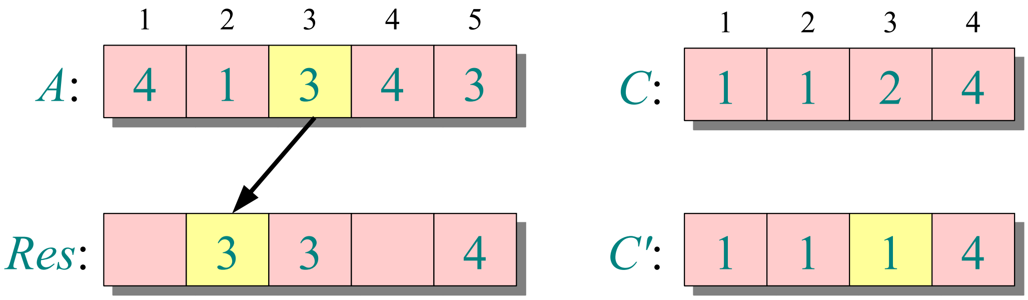
```
for $j \leftarrow n$ downto $1$ do
    $Res[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4



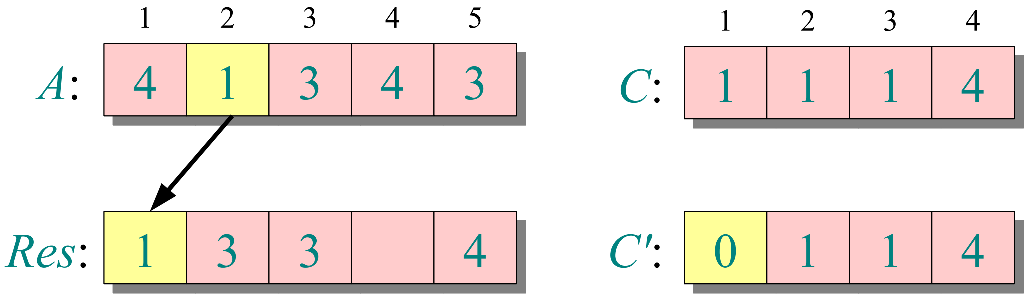
```
for $j \leftarrow n$ downto $1$ do
    $Res[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4



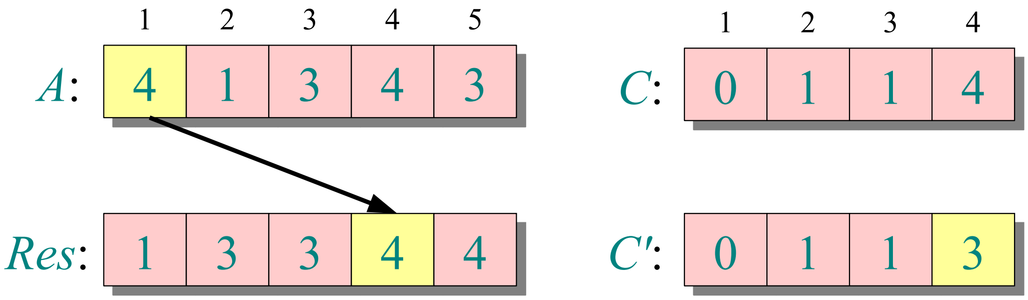
```
for $j \leftarrow n$ downto $1$ do
    $Res[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4



```
for $j \leftarrow n$ downto $1$ do
    $Res[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
```

Loop 4



```
for $j \leftarrow n$ downto $1$ do
    $Res[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
```

Analysis

```
$\Theta(k)$   for $i \leftarrow 1$ to $k$ do
                  $C[i] \leftarrow 0$
$\Theta(n)$   for $j \leftarrow 1$ to $n$ do
                  $C[A[j]] \leftarrow C[A[j]] + 1$
$\Theta(k)$   for $i \leftarrow 2$ to $k$ do
                  $C[i] \leftarrow C[i] + C[i-1]$
$\Theta(n)$   for $j \leftarrow n$ downto $1$ do
                  $Res[C[A[j]]] \leftarrow A[j]$
                  $C[A[j]] \leftarrow C[A[j]] - 1$
```

$$\Theta(n + k)$$

Execution time

If $k \in O(n)$ the counting sort takes $\Theta(n)$ time

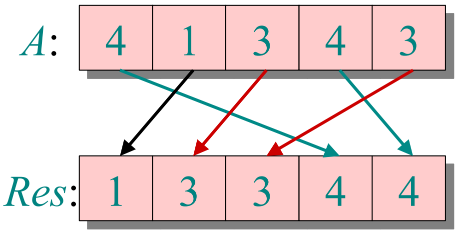
- But sorting takes $\Omega(n \log n)$ time!
- What is wrong?

Answer

- *comparison-based sort* takes $\Omega(n \log n)$ time
- Counting-sort is *not* comparison-based
- In fact, not a single comparison performed between some elements!

Stable sorting

Counting-sort is a stable sorting method: it preserves the input order of equal elements



To think about:
What are the other stable sorting methods?

3.2 Bucket-sort

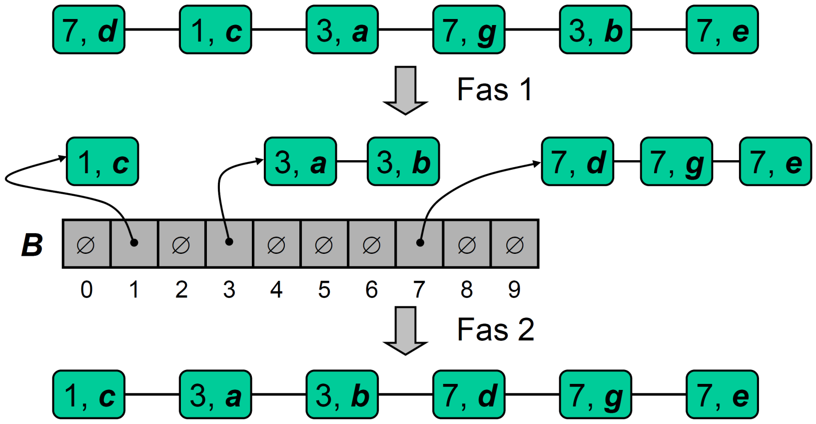
Bucket-sort

- Let S be a sequence of n elements (key, value) with keys from $[0, N-1]$
- Bucket-sort uses the keys as indexes in a help array B of sequences
  - Phase 1: Empty the sequence S by moving each item (k, v) last in its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$ move items in bucket $B[i]$ to the end of the sequence S
- Analysis:
  - Phase 1 runs for $O(n)$ time
  - Phase 2 runs for $O(n+N)$ time

  Bucket-sort runs for $O(n+N)$ time

```
procedure BUCKETSORT($S, N$)
    $B \leftarrow$ array with $N$ empty sequences
    while $\neg S$.ISEMPTY() do
        $f \leftarrow S$.FIRST()
        $(k, o) \leftarrow S$.REMOVE($f$)
        $B[k]$.INSERTLAST($(k, o)$)
    for $i \leftarrow 0$ to $N-1$ do
        while $\neg B[i]$.ISEMPTY() do
            $f \leftarrow B[i]$.FIRST()
            $(k, o) \leftarrow B[i]$.REMOVE($f$)
            $S$.INSERTLAST($(k, o)$)
```

Example: Keys from $[0, 9]$

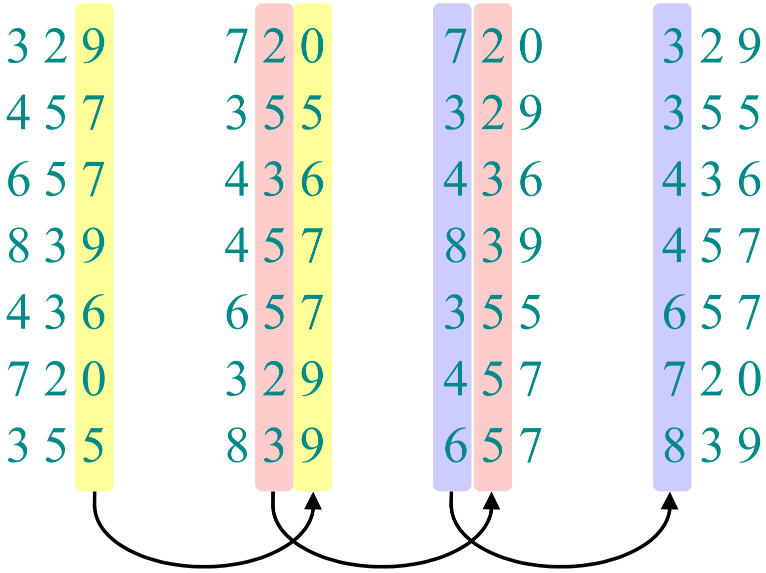


3.3 Radix-sort

Radix-sort

- Origin: Herman Holleriths card sorting machine for census 1890 in USA
- Holleriths original idea: sort the most significant digit first
- Good idea: sort of *least significant digits first* with an external *stable* sorting routine

Example: Execution of radix-sort

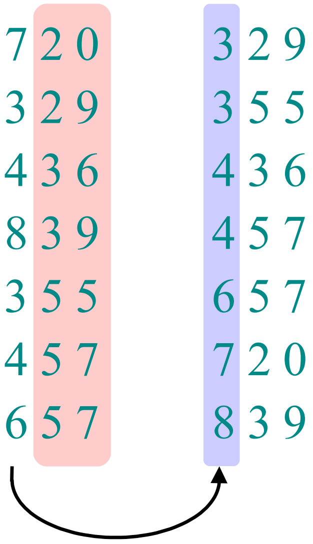


3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Correctness of radix-sort

Use of induction on the digits position

- Suppose that the numbers are sorted on their $t-1$ lowest digits
- Sort based on digit t

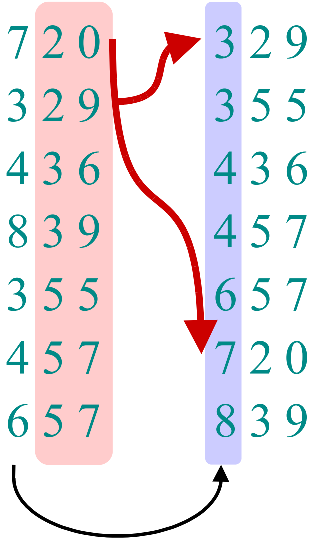


7 2 0	3 2 9
3 2 9	3 5 5
4 3 6	4 3 6
8 3 9	4 5 7
3 5 5	6 5 7
4 5 7	7 2 0
6 5 7	8 3 9

Correctness of radix-sort

Use of induction on the digits position

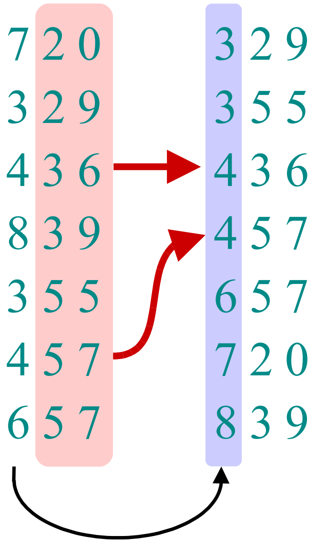
- Suppose that the numbers are sorted on their $t-1$ lowest digits
- Sort based on digit t
  - Two numbers that differ in the number t is correctly sorted



Correctness of radix-sort

Use of induction on the digits position

- Suppose that the numbers are sorted on their $t-1$ lowest digits
- Sort based on digit t
  - Two numbers that differ in the number t are correctly sorted
  - Two numbers that are equal in number t get the same order as in the input data $\Rightarrow$ right order

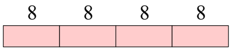


Analysis of radix-sort

- Suppose the counting-sort is used as an external sorting routine
- Sort n machine word on b bits each
- We can see that every word has b/r characters in base 2^r

Example:

32-bit words



$r = 8 \Rightarrow b/r = 4$ pass of counting-sort on digits in base 2^8
or $r = 16 \Rightarrow b/r = 2$ pass of counting-sort on digits in base 2^{16}

How many pass we should do?

Analysis of radix-sort

Remember: counting-sort runs for $\Theta(n+k)$ time to sort n numbers from $[0, k-1]$. If every b-bit word is broken up into r-bit pieces, each takes pass of the counting-sort takes $\Theta(n+2^r)$. since there are b/r pass we get

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Choose r to minimize $T(n,b)$

- Raising r with few passes, but when $r \gg \log n$ time increases exponentially.

21.66

Choosing r

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Minimizing $T(n,b)$ by differentiate and set it to 0. Or, note that we do not want to have $2^r \gg n$, it does not harm asymptotically to choose R as large as possible given the conditions. The choice $r = \log n$ means $T(n,b) = \Theta(bn/\log n)$.

- For a number in the interval 0 to $n^d - 1$ we get $b = d \log n \Rightarrow$ radix-sort runs in $\Theta(dn)$ time.

21.67

Conclusions

In practice, radix-sort is fast for large inputs, as well as easy to code and maintain.

Example: 32-bits number

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge-sort and quick-sort use at least $\lceil \log 2000 \rceil = 11$ pass.

Drawback: It is not possible to sort in-place the counting sort.

21.68