Round 2: Sorting

Tommi Junntila

Aalto University
School of Science
Department of Computer Science

CS-A1140 Data Structures and Algorithms
Autumn 2017
Material

In *Introduction to Algorithms, 3rd ed.* (online via Aalto lib):

- Insertion sort: Sections 2.1 and 2.2
- Mergesort: Section 2.3
- Quicksort: Chapter 7
- Lower bounds for sorting: Section 8.1
- Sorting in linear time: Section 8.2–8.4

Other sources:

- Chapter 2 in *Algorithms, 4th ed.*
- OpenDSA

Note

In this round we will introduce few of the most common sorting algorithms. In addition to these, several others exist — this [wikipedia page](https://en.wikipedia.org) contains many others and also a comparison of several algorithms with respect to the required run-time, extra space etc.
External links:

- MIT 6.006 OCW video on insertion and merge sorts
- MIT 6.006 OCW video on counting sort, radix sort and lower bounds for sorting and searching
- MIT OCW video on Quicksort, Randomized Algorithms
- MIT 6.046J 2015 OCW video on Randomized Select and Randomized Quicksort
- 15 Sorting Algorithms in 6 Minutes video
- www.sorting-algorithms.com and links there
- visualizations of comparison based sorting algorithms
- http://aarondufour.com/tools/visualizer/
Motivation

- Sorting is an ubiquitous task: ordering students by id, web access logs by host, ...
- It is also used as a subroutine in many other algorithms
  - binary search
  - removing duplicates
  - graph search algorithms
  - ...
- With proper algorithms, sorting can be done very efficiently
  Almost as fast as just reading the input
The setting

- We shall sort a sequence (array) of objects.
- Objects are be ordered by some ordering relation \( \leq \).
- Non-primitive data types: the sequence contains pointers/references to the objects, swaps are (relatively) cheap.

Comparisons can be expensive for non-primitive data types.
Possible efficiency aspects: time-complexity, number of array accesses, number of swaps, number of comparisons.
Structured objects can also be sorted with respect to some specific field or substructure called *key*.

For instance, below the employees are sorted by their age — note that two distinct objects can be considered equal with respect to the key.

In the following, we usually only consider arrays of integers in the examples, generalization to arbitrary elements and keys easy.
“In place” sorting

Definition
A sorting algorithm works in place if only a constant amount of the array elements are stored outside the array at any moment of time.

- Thus in place sorting algorithms do not need lots of additional memory
  - this can be important when sorting very large arrays or when the amount of available memory is limited (embedded systems etc)
- On the other hand, many sorting algorithms that are not in place use an auxiliary array of size $\Theta(n)$
The definition above is quite liberal, allowing, for instance, the use of a linear amount of array indices in the recursive call stack of the algorithm.

Stricter versions do exist, see e.g. this wikipedia-page for discussion.

More generally, we could consider the amount of auxiliary space required by an algorithm. It is the amount of space that the algorithm may use in addition to its input.

In this round we assume that the input arrays can be written as well so that the algorithms do not necessarily have to always allocate a new array for the resulting sorted array.
Stability

Definition

A sorting algorithm is stable if it keeps the relative order of equal elements intact.

- In Scala 2.11.8, the `sorted` and `sortByWith` methods of sequences\(^1\) are stable. In addition, `sortBy` is implemented by using `sorted` and is also stable.

- For instance, we can sort

  ```scala
  val l = List(("c",2),("b",3),("e",2),("a",3),("b",2),("a",2))
  ```

  to lexicographical order (first field more significant) by first

  ```scala
  scala> val tmp = l.sortBy(_.2)
  tmp = List((c,2), (e,2), (b,2), (a,2), (b,3), (a,3))
  ```

  and then

  ```scala
  scala> val result = tmp.sortBy(_.1)
  result = List((a,2), (a,3), (b,2), (b,3), (c,2), (e,2))
  ```

  as stable sorting does not put, e.g., `(a,3)` before `(a,2)` even though their first fields are the same.

\(^1\)those that implement `SeqLike`, such as `List` and `Array`
The **algorithm library** in C++11 includes separate functions implementing
- stable sorting (the `stable_sort` function), and
- sorting that is not guaranteed to be stable (the `sort` function)

The latter has a slightly better asymptotic worst-case performance when there is not enough memory for an auxiliary array available.
Insertion sort
A very simple sorting algorithm
Efficient only on small arrays
The idea:
▶ Assume that the first $i$ elements in the array are already sorted (In the first round, $i = 1$)
▶ Take the $i+1$:th element and find the position $j$ where it should belong in the sorted array prefix
▶ Shift the elements $j+1, \ldots, i$ one step right
▶ Insert the element in the $j$th position
▶ Now the first $i + 1$ elements are sorted
▶ Repeat until the whole array is sorted

Example

The input array $a = [a_0, a_1, a_2, a_3]$
A very simple sorting algorithm
Efficient only on small arrays
The idea:
▶ Assume that the first $i$ elements in the array are already sorted
  (In the first round, $i = 1$)
▶ Take the $i + 1$:th element and find the position $j$ where it should belong in the sorted array prefix
▶ Shift the elements $j + 1, \ldots, i$ one step right
▶ Insert the element in the $j$th position
▶ Now the first $i + 1$ elements are sorted
▶ Repeat until the whole array is sorted

Example

In the beginning of the first round, the sub-array $[a_0]$ is sorted, shown in blue

Shift the next element 17 at index 1 leftwards until it is in the right position

I.e., shift 23 to index 1 as $23 > 17$ and move 17 to index 0
- A very simple sorting algorithm
- Efficient only on small arrays
- The idea:
  - Assume that the first $i$ elements in the array are already sorted (In the first round, $i = 1$)
  - Take the $i+1$:th element and find the position $j$ where it should belong in the sorted array prefix
  - Shift the elements $j+1, \ldots, i$ one step right
  - Insert the element in the $j$:th position
  - Now the first $i+1$ elements are sorted
  - Repeat until the whole array is sorted

Example

- In the beginning of the second round, the sub-array $[a_0, a_1]$ is sorted
- Shift the next element 25 at index 2 leftwards until it is in the right position
- I.e., don’t move anything as $23 \leq 25$
A very simple sorting algorithm
Efficient only on small arrays
The idea:

- Assume that the first $i$ elements in the array are already sorted
  (In the first round, $i = 1$)
- Take the $i+1$:th element and find the position $j$ where it should belong in
  the sorted array prefix
- Shift the elements $j+1, \ldots, i$ one step right
- Insert the element in the $j$th position
- Now the first $i+1$ elements are sorted
- Repeat until the whole array is sorted

Example

- In the beginning of the third round, the sub-array
  $[a_0, a_1, a_2]$ is sorted
- Shift the next element 18 at index 3 leftwards until
  it is in the right position
- I.e., shift 25 to index 3 as $25 > 18$, shift 23 to
  index 2 as $23 > 18$, and move 18 to index 1
A very simple sorting algorithm
Efficient only on small arrays
The idea:
▶ Assume that the first $i$ elements in the array are already sorted
   (In the first round, $i = 1$)
▶ Take the $i + 1$:th element and find the position $j$ where it should belong in
   the sorted array prefix
▶ Shift the elements $j + 1, \ldots, i$ one step right
▶ Insert the element in the $j$:th position
▶ Now the first $i + 1$ elements are sorted
▶ Repeat until the whole array is sorted

Example
▶ The array is now sorted
In pseudocode:

Insertion-sort($A$):

```plaintext
for $i \leftarrow 1$ until $A$.length:
    $e \leftarrow A[i]$ // element to be moved
    $j \leftarrow i$
    while $j > 0$ and $A[j-1] > e$:
        $A[j] \leftarrow A[j-1]$
        $j \leftarrow j - 1$
    $A[j] \leftarrow e$
```

Note: unlike in Introduction to Algorithms, 3rd ed. (online via Aalto lib), we start array indexing from 0
An implementation for integer arrays in Scala:

```scala
def insertionSort(a: Array[Int]): Unit = {
  var i = 1
  while (i < a.length) {
    val key = a(i)
    var j = i;
    while (j > 0 && a(j - 1) > key) {
      a(j) = a(j - 1)
      j -= 1
    }
    a(j) = key
    i += 1
  }
}
```
Properties of the algorithm:

- **In place** because only the element currently moved is stored outside the array at any point of time
- **Stable** because we find the rightmost position where the element belongs to, meaning that it is never shifted left past an equal element
Analysis

- In the best case, $\Theta(n)$ run-time
  To see this, consider how the algorithm behaves on arrays that are already sorted

- In the worst case, $\Theta(n^2)$ run-time
  Consider an array of $n$ distinct elements that is sorted in the reverse order
    - the second element is moved 1 steps
    - the third element is moved 2 steps
    - ...
    - the $n$:th element is moved $n - 1$ steps
  Each move of $i$ steps induces the same amount of shifts $\Rightarrow$ at $i$:th round one performs $\Theta(i)$ comparisons and array accesses

$$T(n) = \Theta(n) + T(n - 1) = \Theta(n^2)$$

- The same best and worst-case limits hold for the number of array accesses and comparisons as well
What about average case performance?

- Consider an array consisting of \( n \) distinct random numbers
- At each iteration when moving the \( i \)th element \( a_i \) to its right position, there are on average \( i/2 \) elements in the sub-array \([a_0, \ldots, a_{i-1}]\) that are smaller than \( a_i \)
  \[ \Rightarrow \text{the work on average is } \sum_{i=1}^{n-1} \frac{i}{2} = \Theta(n^2) \]

Recall: in these analyses it is assumed that both

- the comparison of two elements, as well as
- the array accesses

are constant-time operations
How about the performance on a specific application?

- Depends on the inputs the application provides
- If the input is usually almost sorted already, insertion sort may work fine in practice
- If the inputs are usually in random order or in order causing $O(n)$ elements to be moved long distances, we should probably consider some other algorithm
- Note: insertion sort code is very “compact” (small constants) and the memory accesses are very local
  - $\Rightarrow$ for small arrays, insertion sort can be very competitive

A possible optimization: if the comparisons are expensive, use binary search for finding the correct place for insertion

- The number of comparisons drops to $O(n \log_2 n)$ in the worst case
- But the overall run-time stays $\Theta(n^2)$ as the shifts still need to be done
Generics vs fixed types

- A generic insertion sort for any comparable element type:

```scala
def insertionSort[A <% Ordered[A]](a: Array[A]) = {
  var i = 1
  while (i < a.length) {
    val key = a(i)
    var j = i;
    while (j > 0 && a(j-1) > key) {
      a(j) = a(j-1)
      j -= 1
    }
    a(j) = key
    i += 1
  }
}
```

- Unfortunately, with current scala compilers, the generality does not come without a severe performance penalty (see the next slide)
Applying insertion sort to arrays of random integers

To apply the comparison method in the `Ordered` trait, each integer fetched from the array is “boxed”, i.e., wrapped into an object, causing a memory allocation and then later garbage collection.
Mergesort
Mergesort is a commonly used, important sorting algorithm.

It has guaranteed $O(n \log_2 n)$ worst-case run-time.

The high-level idea:

- If we have two sorted arrays, it is easy to *merge* them to a sorted array containing the elements in both arrays.
- Therefore, we can:
  - recursively divide the input array into smaller consecutive sub-arrays until they are easy to sort (e.g., until they have size 1), and
  - then merge the results into longer and longer sorted sub-arrays until the whole array is sorted.

Mergesort thus follows the very important and widely applied *divide and conquer* algorithm design paradigm:

- **Divide**: The problem is recursively divided into smaller subproblems until they become easy enough.
- **Conquer**: the small subproblems are solved.
- **Combine**: The results of the subproblems are then combined to give a solution to the original problem.
Example

Mergesorting a small array, the high-level idea:

original

\[ \begin{array}{cccccc}
21 & 17 & 19 & 1 & 21 & 7 \\
7 & 2 & 20 & & & \\
\end{array} \]

divide

\[ \begin{array}{cccccc}
21 & 17 & 19 & 1 & 21 & 7 \\
7 & 2 & 20 & & & \\
\end{array} \]

divide

\[ \begin{array}{cccccc}
21 & 17 & 19 & 1 & 21 & 7 \\
7 & 2 & 20 & & & \\
\end{array} \]

divide

\[ \begin{array}{cccccc}
21 & 17 & 19 & 1 & 21 & 7 \\
7 & 2 & 20 & & & \\
\end{array} \]

divide

\[ \begin{array}{cccccc}
21 & 17 & 19 & 1 & 21 & 7 \\
7 & 2 & 20 & & & \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]

merge

\[ \begin{array}{cccccc}
1 & 2 & 7 & 17 & 19 & 20 & 21 \\
\end{array} \]
Implementing the merge operation

- In the division phase, we split a sub-array into two consecutive sub-arrays. There is no need to copy the sub-arrays anywhere but maintaining start and end indices is enough.
- In the merge phase, the sorted sub-arrays to be merged are also consecutive.
- To make the larger merged sub-array, we first merge the sorted sub-arrays in an auxiliary array and the copy the result back.
- The merging maintains two indices, $i$ and $j$, in the smaller sub-arrays and at each step appends the smaller value to the aux array and increases to index in question.

Example: Merging two consecutive, sorted sub-arrays

- Suppose that we have already sorted the consecutive sub-arrays $[\text{start}, \text{mid}-1]=[0, 1]$ and $[\text{mid}, \text{end}]=[2, 3]$. 

```
original 17 21 1 19 21 7 2 20
aux      
```

```
start  
mid   
end   
dest
```
Implementing the merge operation

- In the division phase, we split a sub-array into two consecutive sub-arrays. There is no need to copy the sub-arrays anywhere but maintaining start and end indices is enough.
- In the merge phase, the sorted sub-arrays to be merged are also consecutive.
- To make the larger merged sub-array, we first merge the sorted sub-arrays in an auxiliary array and copy the result back.
- The merging maintains two indices, \( i \) and \( j \), in the smaller sub-arrays and at each step appends the smaller value to the aux array and increases the index in question.

Example: Merging two consecutive, sorted sub-arrays

- In the first step, we compare 17 and 1.
- 1 is copied to the aux array.
Implementing the merge operation

- In the division phase, we split a sub-array into two consecutive sub-arrays. There is no need to copy the sub-arrays anywhere but maintaining start and end indices is enough.

- In the merge phase, the sorted sub-arrays to be merged are also consecutive.

- To make the larger merged sub-array, we first merge the sorted sub-arrays in an auxiliary array and then copy the result back.

- The merging maintains two indices, \( i \) and \( j \), in the smaller sub-arrays and at each step appends the smaller value to the aux array and increases to index in question.

Example: Merging two consecutive, sorted sub-arrays

- Next, 17 is copied to the aux array.
Implementing the merge operation

- In the division phase, we split a sub-array into two consecutive sub-arrays. There is no need to copy the sub-arrays anywhere but maintaining start and end indices is enough.
- In the merge phase, the sorted sub-arrays to be merged are also consecutive.
- To make the larger merged sub-array, we first merge the sorted sub-arrays in an auxiliary array and the copy the result back.
- The merging maintains two indices, \( i \) and \( j \), in the smaller sub-arrays and at each step appends the smaller value to the aux array and increases to index in question.

**Example: Merging two consecutive, sorted sub-arrays**

- Then 19 is copied to the aux array.
Implementing the merge operation

- In the division phase, we split a sub-array into two consecutive sub-arrays. There is no need to copy the sub-arrays anywhere but maintaining start and end indices is enough.
- In the merge phase, the sorted sub-arrays to be merged are also consecutive
- To make the larger merged sub-array, we first merge the sorted sub-arrays in an auxiliary array and the copy the result back
- The merging maintains two indices, \( i \) and \( j \), in the smaller sub-arrays and at each step appends the smaller value to the aux array and increases to index in question

Example: Merging two consecutive, sorted sub-arrays

- Now the right sub-array index \( j \) is out of the sub-array
- Copy the rest of the left sub-array to the aux array
Implementing the merge operation

- In the division phase, we split a sub-array into two consecutive sub-arrays. There is no need to copy the sub-arrays anywhere but maintaining start and end indices is enough.
- In the merge phase, the sorted sub-arrays to be merged are also consecutive.
- To make the larger merged sub-array, we first merge the sorted sub-arrays in an auxiliary array and the copy the result back.
- The merging maintains two indices, \( i \) and \( j \), in the smaller sub-arrays and at each step appends the smaller value to the aux array and increases to index in question.

Example: Merging two consecutive, sorted sub-arrays

Finally, copy the merged sub-array back to the original array.
An implementation of merge in Scala:

```scala
def merge(a: Array[Int], aux: Array[Int],
           start: Int, mid: Int, end: Int): Unit = {
  var (i, j, dest) = (start, mid, start)
  while (i < mid && j <= end) {
    // Merge to aux, smallest first
    if (a(i) <= a(j)) {
      aux(dest) = a(i); i += 1
    } else {
      aux(dest) = a(j); j += 1
    }
    dest += 1
  }
  while (i < mid) {
    aux(dest) = a(i); i += 1; dest += 1
  } // Copy rest
  while (j <= end) {
    aux(dest) = a(j); j += 1; dest += 1
  } // Copy rest
  dest = start // Copy from aux back to a
  while (dest <= end) {
    a(dest) = aux(dest); dest += 1
  }
}
```

The run-time of merge is $\Theta(k)$, where $k$ is the size of the two consecutive subarrays
Once we have the merge routine, the recursive mergesort algorithm itself is rather simple

```scala
def mergesort(a: Array[Int]): Unit = {
  if (a.length <= 1) return

  // Auxiliary memory for doing merges
  val aux = new Array[Int](a.length)

  // The recursive main algorithm
  def _mergesort(start: Int, end: Int): Unit = {
    if (start >= end)
      return // One or zero elements, do nothing
    val leftEnd = start + (end - start) / 2 // The midpoint
    _mergesort(start, leftEnd) // Sort the left half
    _mergesort(leftEnd + 1, end) // Sort the right half
    merge(a, aux, start, leftEnd + 1, end) // Merge the results
  }
  _mergesort(0, a.length - 1)
}
```

Above for integers arrays, extension to other element types as well as to generic `Ordered` trait straightforward
Time-complexity

- The recurrence for the run-time of merge sort is

\[ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \]

where \( c \) is a positive constant.

- If \( n \) is a power of two, we can substitute the parameter with \( n = 2^k \) and get

\[ T(2^k) = T(2^{k-1}) + T(2^{k-1}) + c2^k = c2^k + 2T(2^{k-1}) \]

- Expanding the recurrence, we obtain

\[
\begin{align*}
T(2^k) &= c2^k + 2(c2^{k-1} + 2T(2^{k-2})) \\
&= c2^k + 2c2^{k-1} + 4(c2^{k-2} + 2T(2^{k-3})) \\
&= \ldots \\
&= kc2^k
\end{align*}
\]

- Recalling that \( k = \log_2 n \), we get \( T(n) = cn\log_2 n = \Theta(n\log_2 n) \)
We can also analyze the required run-time graphically by showing the computation of $T(n)$ as a tree:

Here we omit the floors and ceilings and use the recurrence

$$T(1) = c \text{ and } T(n) = T(n/2) + T(n/2) + cn$$

The red annotations beside nodes illustrate the work done at the node itself.

The recursion stops when $\frac{n}{2^i} \leq 1$, i.e. when $i \geq \log_2 n$, and thus the tree has $\log_2 n$ levels.

Total time at level

Total in all levels: $nc \log_2 n$
An experimental evaluation

- For larger arrays, merge sort performs significantly better than insertion sort and scales better as well.
On arrays with random integers, insertion sort is competitive only for very small arrays.
Handling small sub-arrays

- Each recursive call induces a function invocation ...
- and function invocations involve some instructions and memory activity (getting and initializing new stack frame)
- Observing the relative efficiency of insertion sort on small sub-arrays illustrated in the evaluation above, we consider a variant in which recursion is not performed all the way to unit sized sub-arrays but insertion sort is called for sub-arrays whose size is below some predetermined threshold value
The modification in Scala

```scala
def mergesort(a: Array[Int], threshold: Int = 64): Unit = {
  if (a.length <= 1) return
  val aux = new Array[Int](a.length)
  def _mergesort(start: Int, end: Int): Unit = {
    if (end - start < threshold) insertionsort(a, start, end) // Changed
    else {
      val leftEnd = (start + end) / 2
      _mergesort(start, leftEnd)
      _mergesort(leftEnd + 1, end)
      merge(a, aux, start, leftEnd + 1, end)
    }
  }
  _mergesort(0, a.length - 1)
}
```

Above, `insertionsort(A, start, end)` sorts the subarray `A[start, end]` with insertion sort
A decent constant-factor speedup is achieved
Quicksort
Another widely used sorting algorithm
Like mergesort, quicksort is a divide and conquer algorithm
Easy to make work in place
The basic idea for sorting a sub-array \(A[lo, hi]\) with \(lo \leq hi\):

1. **Base case:** if \(lo = hi\), the sub-array \(A[lo, hi]\) has only one element and is thus sorted
2. **Divide:** select a *pivot* element \(p\) in \(A[lo, hi]\), and *partition* \(A[lo, hi]\) in three sub-arrays \(A[lo, q - 1]\), \(A[q, q]\) and \(A[q + 1, hi]\) such that
   - all the elements in \(A[lo, q - 1]\) are less than or equal to \(p\),
   - \(A[q, q] = [p]\), and
   - all the elements in \(A[q + 1, hi]\) are greater than \(p\)

After this step, the element \(p\) is in the right place
3. **Conquer:** recursively sort the sub-arrays \(A[lo, q - 1]\) and \(A[q + 1, hi]\) with quicksort

The main recursive function in Scala

```scala
def quicksort(a: Array[Int]): Unit = {
  def _quicksort(lo: Int, hi: Int): Unit = {
    val q = partition(a, lo, hi)
    if (lo < q - 1) _quicksort(lo, q - 1)
    if (q + 1 < hi) _quicksort(q + 1, hi)
  }
  if (a.length >= 2) _quicksort(0, a.length - 1)
}
```
Example: sorting an array with quicksort

- The **Scala** program in the previous slide
- The recursive calls and results after partitioning in the order they are executed
- green shows the sub-array $A[lo, hi]$
- red highlights the pivot element
- blue gives the left sub-array after partitioning
- cyan gives the right sub-array after partitioning

<table>
<thead>
<tr>
<th>Array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>after partition(0,7)</td>
<td>17</td>
<td>19</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>20</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>after partition(2,4)</td>
<td>1</td>
<td>2</td>
<td>17</td>
<td>7</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>after partition(6,7)</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>
The same in a tree form, showing that (unlike in the case of mergesort) the recursion tree is not very well balanced.
A partitioning algorithm (from *Introduction to Algorithms, 3rd ed.* (online via Aalto lib)):

- Given a sub-array $A[lo, hi]$
- Select the pivot element $p$ in it
- Swap the pivot element with the last one so that it is at the end of the sub-array
- Starting from $l$, maintain two indices $i$ and $j$ such that
  - elements left of $i$ in the sub-array are at most $p$
  - elements from $i$ until $j$ are larger than $p$
- Move the index $j$ rightwards and if the element at it is at most $p$, swap the elements at $i$ and $j$ and increase $i$ by one
- Finally, swap the last element (the pivot) with the one in index $i$

**Example**

- The initial situation
- The pivot element, in red, at the end
- $21 > 20$, increase $j$
A partitioning algorithm (from Introduction to Algorithms, 3rd ed. (online via Aalto lib)):

- Given a sub-array $A[lo, hi]$
- Select the pivot element $p$ in it
- Swap the pivot element with the last one so that it is at the end of the sub-array
- Starting from $l$, maintain two indices $i$ and $j$ such that
  - elements left of $i$ in the sub-array are at most $p$
  - elements from $i$ until $j$ are larger than $p$
- Move the index $j$ rightwards and if the element at it is at most $p$, swap the elements at $i$ and $j$ and increase $i$ by one
- Finally, swap the last element (the pivot) with the one in index $i$

Example

- The elements from $i$ until $j$ are greater than the pivot and in cyan
- $17 \leq 20$, swap elements at $i$ and $j$
- Increase $i$ and $j$
A partitioning algorithm (from Introduction to Algorithms, 3rd ed. (online via Aalto lib)):

- Given a sub-array $A[lo, hi]$
- Select the pivot element $p$ in it
- Swap the pivot element with the last one so that it is at the end of the sub-array
- Starting from $i$, maintain two indices $i$ and $j$ such that
  - elements left of $i$ in the sub-array are at most $p$
  - elements from $i$ until $j$ are larger than $p$
- Move the index $j$ rightwards and if the element at it is at most $p$, swap the elements at $i$ and $j$ and increase $i$ by one
- Finally, swap the last element (the pivot) with the one in index $i$

Example

- The elements from left until $i$ are less or equal to the pivot and in blue
- $19 \leq 20$, swap elements at $i$ and $j$
- Increase $i$ and $j$
A partitioning algorithm (from Introduction to Algorithms, 3rd ed. (online via Aalto lib)):

- Given a sub-array $A[lo, hi]$
- Select the pivot element $p$ in it
- Swap the pivot element with the last one so that it is at the end of the sub-array
- Starting from $l$, maintain two indices $i$ and $j$ such that
  - elements left of $i$ in the sub-array are at most $p$
  - elements from $i$ until $j$ are larger than $p$
- Move the index $j$ rightwards and if the element at it is at most $p$, swap the elements at $i$ and $j$ and increase $i$ by one
- Finally, swap the last element (the pivot) with the one in index $i$

**Example**

- The elements from left until $i$ are less or equal to the pivot and in blue
- $1 \leq 20$, swap elements at $i$ and $j$
- Increase $i$ and $j$
A partitioning algorithm (from Introduction to Algorithms, 3rd ed. (online via Aalto lib)):

- Given a sub-array \( A[lo, hi] \)
- Select the pivot element \( p \) in it
- Swap the pivot element with the last one so that it is at the end of the sub-array
- Starting from \( l \), maintain two indices \( i \) and \( j \) such that
  - elements left of \( i \) in the sub-array are at most \( p \)
  - elements from \( i \) until \( j \) are larger than \( p \)
- Move the index \( j \) rightwards and if the element at it is at most \( p \), swap the elements at \( i \) and \( j \) and increase \( i \) by one
- Finally, swap the last element (the pivot) with the one in index \( i \)

**Example**

- Continue until \( j \) is at the right end
A partitioning algorithm (from Introduction to Algorithms, 3rd ed. (online via Aalto lib)):

- Given a sub-array \( A[lo, hi] \)
- Select the pivot element \( p \) in it
- Swap the pivot element with the last one so that it is at the end of the sub-array
- Starting from \( l \), maintain two indices \( i \) and \( j \) such that
  - elements left of \( i \) in the sub-array are at most \( p \)
  - elements from \( i \) until \( j \) are larger than \( p \)
- Move the index \( j \) rightwards and if the element at it is at most \( p \), swap the elements at \( i \) and \( j \) and increase \( i \) by one
- Finally, swap the last element (the pivot) with the one in index \( i \)

**Example**

- Swap the last element (the pivot) with the element at \( i \)
- Done
In Scala (based on the version in *Introduction to Algorithms, 3rd ed.* (online via Aalto lib))

```scala
def swap(a: Array[Int], i: Int, j: Int): Unit = {
    val t = a(i); a(i) = a(j); a(j) = t
}

def partition(a: Array[Int], lo: Int, hi: Int): Int = {
    val pivot = a(hi)  // Very simple pivot selection!
    var i = lo - 1
    var j = lo
    while (j < hi) {
        if (a(j) <= pivot) {
            i += 1; swap(a, i, j)
        }
        j += 1
    }
    swap(a, i + 1, hi)
    i + 1
```
An experimental evaluation

- Merge sort vs quicksort on arrays with random integers:

![Graph showing the comparison between Merge sort and Quicksort](graph.png)
Comparison against Scala’s `Array[Int].sorted` and `java.util.Arrays.sort` on arrays of random integers
Unsurprisingly, the implementation of `java.util.Arrays.sort` in this case is a variant of quicksort (see e.g. this source code of a JVM).

Currently, `Array[Int].sorted` maps integers into objects and uses `java.util.Arrays.sort` for objects which currently implements a “TimSort” algorithm (source code of a JVM).
Time complexity

- Based on the above experimental plots, quicksort seems to be an $O(n \log_2 n)$ algorithm?
- Actually, it is not; in the worst case the basic version takes $\Theta(n^2)$ time.
- Partitioning takes $\Theta(k)$ time, where $k$ is the length of the considered subarray.
- In our basic version, we select the last element in the sub-array to be the pivot element.
- The quadratic-time worst case performance occurs when the array is already sorted (or almost sorted)
  - The pivot element does not move anywhere and we effectively just produce one partition that has size $k - 1$
  - Thus the recurrence is $T(n) = \Theta(n) + T(n-1) = \Theta(n^2)$
Pivot selection

- The quadratic time consumption on already sorted data is not good; in practice is can be quite common that the data is already almost in order.
- Thus *pivot selection* is important!
- Worst-case $O(n \log_2 n)$ run-time is obtained when the pivot is chosen so that the obtained two partitions are of roughly equal size.
- Thus one could choose the pivot to be the *median* of the subarray considered.
  - In principle the median can be computed in linear time (see Section 9.3 in *Introduction to Algorithms, 3rd ed.* (online via Aalto lib)) but the constant factors in this operation are quite large.
  - A practical approximation is to select the pivot to be the median of three random elements in the subarray.
An another common and easy strategy is to select a random element in the subarray to be the pivot element

- After the selection, move it to be the last element and proceed as earlier
- For arrays with \( n \) distinct elements, the expected run-time of quicksort is \( O(n \log_2 n) \) when random pivots are chosen (see Section 7.4 in Introduction to Algorithms, 3rd ed. (online via Aalto lib))
- This is what happened in our experiments above: we did not choose random pivots but the elements were random and thus the situation was effectively the same
Multiple equal elements

- Pivot selection does not help when the array has a lot of equal elements
- Consider the case that the array consists of \( n \) elements that are all the same
  - No matter how the pivot is selected, the partitioning of a subarray with \( k \) elements again effectively produces just one partition with \( k - 1 \) elements that are the same as the pivot element
  - Thus the run-time of the basic quicksort algorithm is again \( \Theta(n^2) \)
- This problem can be solved by using a partitioning function that produces three partitions:
  - one with smaller elements,
  - one with all the equal ones (including the pivot element), and
  - one with larger value

\( \text{✍️} \) The partition with equal elements needs not to be further processed but is already sorted

\( \text{✍️} \) implementation left as an exercise
Asymptotic Lower Bounds
How far from optimal is, for instance, mergesort?

To answer this, we must define the computation model we use

- We consider our standard “random access machine” model in which memory locations can be accessed in constant time and in which all memory locations and registers have some fixed bit width
- Thus each step can only access and manipulate a fixed number of bits

Let us further assume a “comparisons-only” restriction in which

1. we can compare two elements $a$ and $b$ with $=, <, \leq, >, \text{ or } \geq$
2. but we cannot obtain any other information, such as their binary encoding, about the elements

For instance, sorting algorithm implementations that only use the Scala Ordered trait interface fall under this assumption.
In the comparison-only model described above, we can get the following asymptotic lower bounds for the numbers of comparisons that any algorithm must make in the worst case:

- Searching an element in an ordered array must make $\Omega(\log_2 n)$ comparisons in the worst-case.
- Thus binary search is, in an asymptotic sense, optimal.
- Sorting an array of $n$ elements requires $\Omega(n\log_2 n)$ comparisons in the worst case.
- Therefore, mergesort is, in an asymptotic sense, optimal.

Proof: see Section 8.1 in Introduction to Algorithms, 3rd ed. (online via Aalto lib)
Counting and Radix Sorts
For some element types, we can do sorting that is not based on direct comparisons

Or in other words, sometimes we can exploit properties of the elements to perform faster sorting

In the following, we consider

- counting sort that is applicable for small element sets, and
- radix sorts that exploit the structure of the element
Counting sort

- Assume that the set of possible elements is \( \{0, \ldots, k - 1\} \) for some reasonably small \( k \)
- For instance, if we wish to sort an array of bytes, \( k = 256 \)
- The idea in counting sort is to
  - first count the occurrences of each element in the array, and
  - then use the cumulated counts to indicate where each element should be sorted to
- Uses \( \Theta(n + k) \) amount of extra space, i.e., not “in place”
Example: Applying counting sort to an array of octal digits

counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length k
result ← int-array of length A.length

for v in 0 until k: count[v] ← 0

// 2: Count occurrences
for i in 0 until A.length:
count[A[i]] += 1

// 3: Cumulative occurrences
cumu ← 0
for v in 0 until k:
current ← count[v]
count[v] ← cumu
cumu += current

// 4: Make the result
for i in 0 until A.length:
result[count[A[i]]] ← A[i]
count[A[i]] += 1

Phase 1

Allocate the auxiliary count and result arrays

Initialize the count array with 0s

Takes time Θ(k)

Here k = 8
Example: Applying counting sort to an array of octal digits

counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length \( k \)
result ← int-array of length \( A.length \)
for \( v \) in 0 until \( k \): \( count[v] \) ← 0
// 2: Count occurrences
for \( i \) in 0 until \( A.length \):
    \( count[A[i]] \) += 1
// 3: Cumulative occurrences
cumu ← 0
for \( v \) in 0 until \( k \):
    current ← \( count[v] \)
    \( count[v] \) ← cumu
    cumu += current
// 4: Make the result
for \( i \) in 0 until \( A.length \):
    result[\( count[A[i]] \)] ← \( A[i] \)
    \( count[A[i]] \) += 1

- Phase 2

- In \( n \) steps count in each position \( count[j] \) how many times the element \( j \) occurs in the input array

- Takes time \( \Theta(n) \)
Example: Applying counting sort to an array of octal digits

counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length k
result ← int-array of length A.length

for v in 0 until k: count[v] ← 0

// 2: Count occurrences
for i in 0 until A.length:
    count[A[i]] += 1

// 3: Cumulative occurrences
cumu ← 0
for v in 0 until k:
    current ← count[v]
    count[v] ← cumu
    cumu += current

// 4: Make the result
for i in 0 until A.length:
    result[count[A[i]]] ← A[i]
    count[A[i]] += 1

- Phase 3

- For each value j, compute
  \[\text{index}[j] = \sum_{i=0}^{j-1} \text{count}[i]\]
  which tells the position of the first occurrence of the element j in the sorted result array

- **Note:** in the pseudocode these values are counted in the count array in order to save space

- Takes time \(\Theta(k)\)
Example: Applying counting sort to an array of octal digits

counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length k
result ← int-array of length A.length

for v in 0 until k: count[v] ← 0

// 2: Count occurrences
for i in 0 until A.length:
    count[A[i]] += 1

// 3: Cumulative occurrences
cumu ← 0
for v in 0 until k:
    current ← count[v]
    count[v] ← cumu
    cumu += current

// 4: Make the result
for i in 0 until A.length:
    result[count[A[i]]] ← A[i]
    count[A[i]] += 1

- Phase 4, first loop iteration
- Copy the element e₀ at index 0 to the index index[e₀] in the result array
- Increment index[e₀] by one
counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length k
result ← int-array of length A.length
for v in 0 until k: count[v] ← 0
// 2: Count occurrences
for i in 0 until A.length:
    count[A[i]] += 1
// 3: Cumulative occurrences
cumu ← 0
for v in 0 until k:
    current ← count[v]
    count[v] ← cumu
    cumu += current
// 4: Make the result
for i in 0 until A.length:
    result[count[A[i]]] ← A[i]
    count[A[i]] += 1

- Phase 4, second loop iteration
- Copy the element e₁ at index 1 to the index index[e₁] in the result array
- Increment index[e₁] by one
Example: Applying counting sort to an array of octal digits

counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length k
result ← int-array of length A.length

for v in 0 until k: count[v] ← 0

// 2: Count occurrences
for i in 0 until A.length:
    count[A[i]] += 1

// 3: Cumulative occurrences
cumu ← 0
for v in 0 until k:
    current ← count[v]
    count[v] ← cumu
    cumu += current

// 4: Make the result
for i in 0 until A.length:
    result[count[A[i]]] ← A[i]
    count[A[i]] += 1

- Phase 4, third loop iteration
- Copy the element e_2 at index 2 to the index index[e_2] in the result array
- Increment index[e_2] by one
Example: Applying counting sort to an array of octal digits

counting-sort(A):

// 1: Allocate and init arrays
count ← int-array of length k
result ← int-array of length A.length
for v in 0 until k: count[v] ← 0
// 2: Count occurrences
for i in 0 until A.length:
    count[A[i]] += 1
// 3: Cumulative occurrences
cumu ← 0
for v in 0 until k:
    current ← count[v]
    count[v] ← cumu
    cumu += current
// 4: Make the result
for i in 0 until A.length:
    result[count[A[i]]] ← A[i]
    count[A[i]] += 1

- Phase 4
- Continue until finished
- Copy the result array back to the original array (if needed)
Properties:

- Stable
- Needs $\Theta(k + n)$ extra memory
- Works in $\Theta(k + n)$ time
- Very good algorithm when $k$ is small
Most-significant-digit-first radix sort

- Assume that
  - the keys are sequences of *digits* from 0, ..., *k* – 1 and
  - we wish to sort them in lexicographical order

- For instance,
  - 32-bit integers are sequences of length 4 of 8-bit bytes
  - ASCII-strings are sequences of ASCII-characters

- The MSD radix sort
  - first sorts the keys according to their first (most significant) digit by using counting sort, and
  - then, recursively, sorts each sub-array with the same most significant digit by considering sequences starting from the second digit

- Properties:
  - Stable
  - Needs $\Theta(k + n)$ extra memory
  - Works in $O((k + n)d)$ time, where $d$ is the maximum length of the sequences
Least-significant-digit-first radix sort

- Again, assume that
  - the keys are sequences of digits from 0, ..., \( k - 1 \) and
  - we wish to sort the in lexicographical order

- But now also assume that all the keys are of the same length \( d \)

- For instance, keys could be
  - 32-bit integers are sequences of length 4 of 8-bit bytes
  - ASCII-strings of the same length, such as Finnish social security numbers

- The LSD radix sort
  - first sorts the keys according to their last (least significant) digit by using the stable counting sort, and
  - then, iteratively, sorts the keys according to their second-last digit by using the stable counting sort and so on.

- Properties:
  - Stable
  - Needs \( \Theta(k + n) \) extra memory
  - Works in \( O((k + n)d) \) time
Example

Sorting 16-bit unsigned (hexadecimal) numbers with LSD radix sort by interpreting them as sequences of four 4-bit digits.
Implementation and experimental evaluation

Left as a programming exercise