Sorting

Tjark Weber
tjark.weber@it.uu.se

Today

- Accumulators
- Sorting Algorithms
  - Insertion Sort
  - Bubble Sort
- Divide and Conquer
  - Merge Sort
  - Quicksort

Accumulators

Accumulators

In functional programming, an **accumulator** is an argument of a (recursive) function that is used to store an intermediate result.

Usually, each recursive call will pass a modified/extended accumulator: the intermediate result changes during the recursive computation.

When the base case is reached, the function returns (a value derived from) the accumulator.

Accumulators: Example

For instance, we can implement the `length` function with an auxiliary function that uses an accumulator:

```
lengthAux acc []     = acc
lengthAux acc (_:xs) = lengthAux (acc+1) xs
```

Note that `lengthAux` solves a more general problem: `lengthAux acc xs` returns `acc + length xs`. Now

```
length xs = lengthAux 0 xs
```

Accumulators: Another Example

We can also implement `reverse` with an auxiliary function that uses an accumulator:

```
reverseAux acc []     = acc
reverseAux acc (x:xs) = reverseAux (x:acc) xs

reverse xs = reverseAux [] xs
```

Note that this implementation is more efficient than the one we saw previously, which called ++ recursively.

Sorting Algorithms

Algorithms

An **algorithm** is any well-defined computational procedure that takes some value (or collection of values) as *input* and produces some value (or collection of values) as *output*.

An algorithm is thus a sequence of computational steps that transform the input into the output.

An algorithm solves a **computational problem**, phrased in terms of a desirable input/output relationship.

Algorithms: Examples

We have already seen many examples of algorithms and problems.

For instance, **Euclid's algorithm** solves the problem of computing the greatest common divisor:

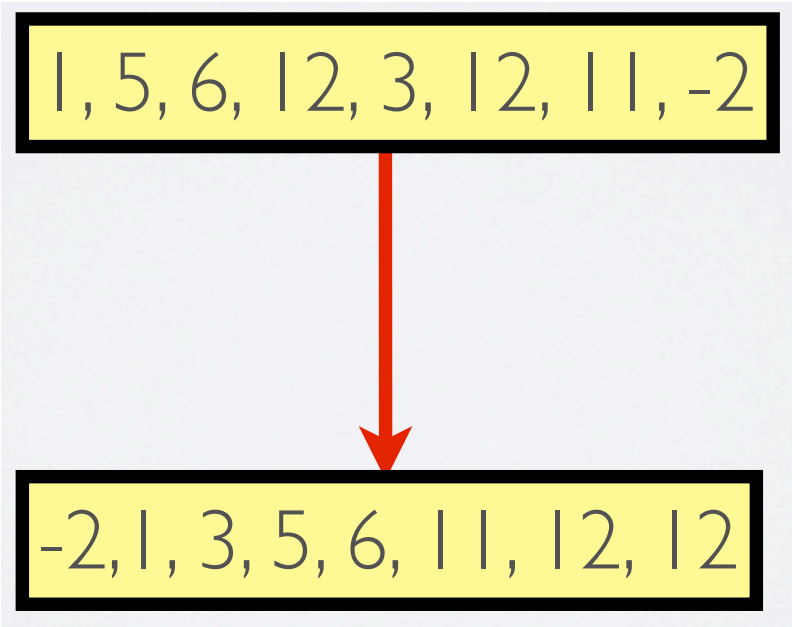
Input: two natural numbers a, b
Output: $\gcd(a, b)$

Often, a problem can be solved by many different algorithms.

The Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

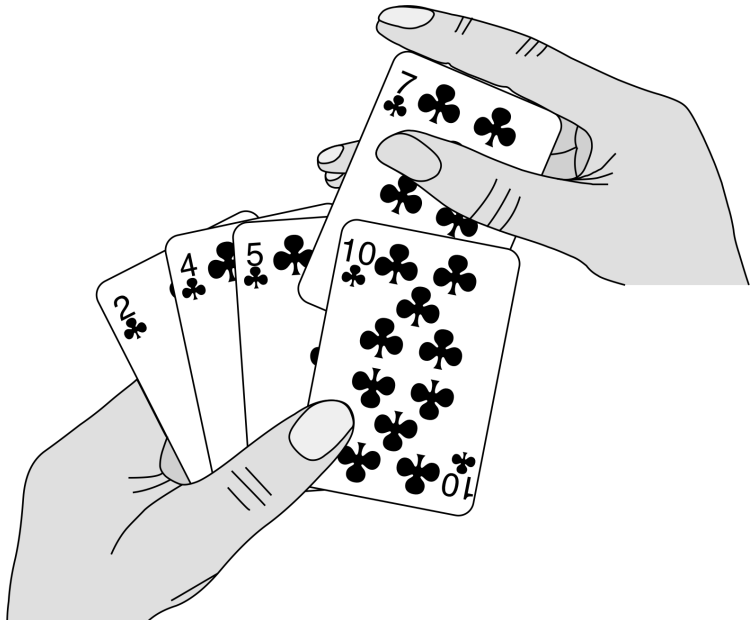


Sorting?

How would you do it?

Insertion Sort

The idea of the insertion sort algorithm is the same as what many people use when sorting a hand of playing cards.



Insertion Sort: Invariant

Insertion sort works with *two* sequences:

- a sorted sequence (the left hand end),
- the (unsorted) remaining part of the input sequence.

The algorithm repeatedly takes an element from the remaining part, and insert this at the proper place into the sorted sequence.

Insertion Sort: Invariant (cont.)

Initially, the sorted sequence is empty, and the remaining part of the input sequence is the entire input sequence.

Sorted	Remaining
[]	[1,5,6,12,3,12,11,-2]

After executing the algorithm, the sorted sequence will contain all input numbers, and the remaining part of the input sequence will be empty.

Sorted	Remaining
[-2,1,3,5,6,11,12,12]	[]

Insertion Sort: Example

Sorted	Remaining
[]	`[1,5,6,12,3,12,11,-2]`

Insertion Sort: Example

Sorted	Remaining
`[]`	`[1,5,6,12,3,12,11,-2]`
`[1]`	`[5,6,12,3,12,11,-2]`

Insertion Sort: Example

Sorted	Remaining
`[]`	`[1,5,6,12,3,12,11,-2]`
`[1]`	`[5,6,12,3,12,11,-2]`
`[1,5]`	`[6,12,3,12,11,-2]`

Insertion Sort: Example

Sorted	Remaining
`[]`	`[1,5,6,12,3,12,11,-2]`
`[1]`	`[5,6,12,3,12,11,-2]`
`[1,5]`	`[6,12,3,12,11,-2]`
`[1,5,6]`	`[12,3,12,11,-2]`

Insertion Sort: Example

Sorted	Remaining
`[]`	`[1,5,6,12,3,12,11,-2]`
`[1]`	`[5,6,12,3,12,11,-2]`
`[1,5]`	`[6,12,3,12,11,-2]`
`[1,5,6]`	`[12,3,12,11,-2]`
`[1,5,6,12]`	`[3,12,11,-2]`

Insertion Sort: Example

Sorted	Remaining
`[]`	`[1,5,6,12,3,12,11,-2]`
`[1]`	`[5,6,12,3,12,11,-2]`
`[1,5]`	`[6,12,3,12,11,-2]`
`[1,5,6]`	`[12,3,12,11,-2]`
`[1,5,6,12]`	`[3,12,11,-2]`
`[1,3,5,6,12]`	`[12,11,-2]`

Insertion Sort: Example

Sorted	Remaining
`[]`	`[1,5,6,12,3,12,11,-2]`
`[1]`	`[5,6,12,3,12,11,-2]`
`[1,5]`	`[6,12,3,12,11,-2]`
`[1,5,6]`	`[12,3,12,11,-2]`
`[1,5,6,12]`	`[3,12,11,-2]`
`[1,3,5,6,12]`	`[12,11,-2]`
`[1,3,5,6,12,12]`	`[11,-2]`

Insertion Sort: Example

Sorted	Remaining
[]	[1,5,6,12,3,12,11,-2]
[1]	[5,6,12,3,12,11,-2]
[1,5]	[6,12,3,12,11,-2]
[1,5,6]	[12,3,12,11,-2]
[1,5,6,12]	[3,12,11,-2]
[1,3,5,6,12]	[12,11,-2]
[1,3,5,6,12,12]	[11,-2]
[1,3,5,6,11,12,12]	[-2]

Insertion Sort: Example

Sorted	Remaining
[]	[1,5,6,12,3,12,11,-2]
[1]	[5,6,12,3,12,11,-2]
[1,5]	[6,12,3,12,11,-2]
[1,5,6]	[12,3,12,11,-2]
[1,5,6,12]	[3,12,11,-2]
[1,3,5,6,12]	[12,11,-2]
[1,3,5,6,12,12]	[11,-2]
[1,3,5,6,11,12,12]	[-2]
[-2,1,3,5,6,11,12,12]	[]

Insertion Sort: Algorithm

If the remaining part of the input sequence is empty then
  return the sorted sequence
else
  remove one element from the remaining part of the input sequence,
  find the correct location where this element belongs within the sorted
    sequence and insert it,
  continue sorting the remaining part of the input sequence.

Insertion Sort: Implementation

Inserting a number k into a sorted sequence, so that the resulting sequence is still sorted:

Insertion Sort: Implementation

Inserting a number k into a sorted sequence, so that the resulting sequence is still sorted:

```
insert k []     = [k]
insert k (x:xs) =
  if k < x then k:x:xs else x:(insert k xs)
```

Insertion sort:

Insertion Sort: Implementation

Inserting a number k into a sorted sequence, so that the resulting sequence is still sorted:

```
insert k []     = [k]
insert k (x:xs) =
  if k < x then k:x:xs else x:(insert k xs)
```

Insertion sort:

```
insertionSortAux sorted [] = sorted
insertionSortAux sorted (x:xs) =
  insertionSortAux (insert x sorted) xs

insertionSort xs = insertionSortAux [] xs
```

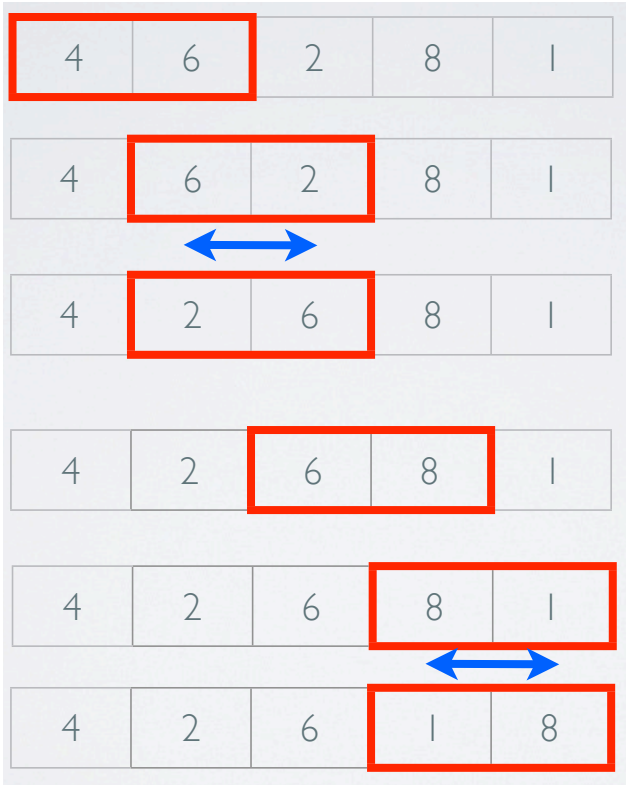
Almost Insertion Sort

```
almostInsertionSort [] = []
almostInsertionSort (h:t) =
  insert h (almostInsertionSort t)
```

Study this code at home. How does it differ from the Insertion Sort algorithm?

Bubble Sort

Scan sequence left to right, swapping elements if in wrong order.



Largest element is now in final position.

Repeat for all but final sorted part.

Almost Bubble Sort

```
bubbleSelect []   = []
bubbleSelect [a] = [a]
bubbleSelect (a:b:t) =
  if a > b then b:(bubbleSelect (a:t))
           else a:(bubbleSelect (b:t))

bubbleSort []     = []
bubbleSort (h:t) = bubbleSelect (h:(bubbleSort t))
```

Study this code at home. How does it differ from the Bubble Sort algorithm?

Divide and Conquer

Divide and Conquer Algorithms

Divide and conquer is a generic approach to solving problems, based on multiple recursion.

The divide-and-conquer strategy solves a problem by

1. *dividing* it into subproblems that are themselves smaller instances of the same type of problem,
2. *recursively solving* (conquering) these subproblems, and
3. appropriately *combining* the solutions for subproblems into a solution for the original problem.

Example: Integer Exponentiation

Remember our algorithm for computing b^n (where $n \in \mathbb{N}$), based on simple recursion:

```
power b 0 = 1
power b n = b * power b (n-1)
```

This is actually fairly inefficient: it requires n recursive calls.

Example: Integer Exponentiation (cont.)

Here is a divide-and-conquer algorithm, based on multiple recursion:

```
power b 0 = 1
power b n
  | odd n = b * power b (n-1)
  | otherwise =
      power b (n `div` 2) * power b (n `div` 2)
```

Example: Integer Exponentiation (cont.)

We can use a local declaration to avoid the repeated computation of `power b (n `div` 2)`:

```
power b 0 = 1
power b n
  | odd n = b * power b (n-1)
  | otherwise =
      let x = power b (n `div` 2) in x * x
```

This algorithm is more efficient (i.e., requires fewer recursive calls) than the algorithm based on simple recursion.

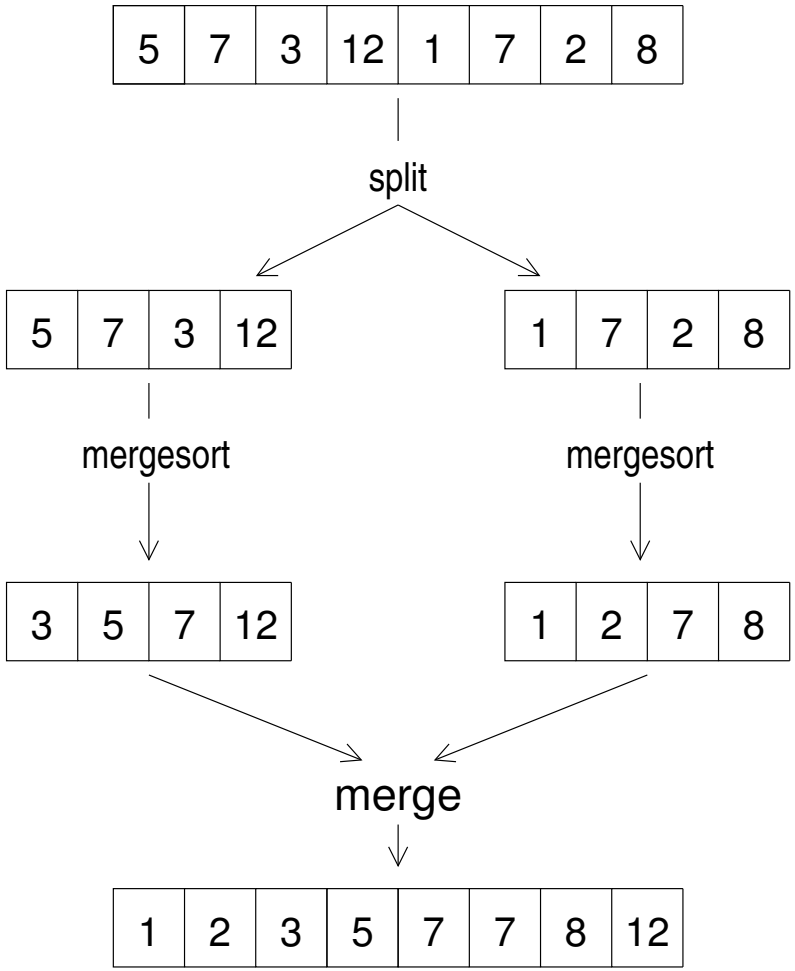
Divide and Conquer: Other Examples

Many well-known, efficient algorithms in computer science are based on the divide-and-conquer strategy:

- merge sort & quicksort
- integer multiplication (Karatsuba, 1962)
- matrix multiplication (Strassen, 1969)
- ...

Merge Sort (John von Neumann, 1945)

Divide & conquer (then combine results)



Merge Sort Implementation: Divide

Splitting a list into two ‘halves’ (naive implementation):

Merge Sort Implementation: Divide

Splitting a list into two 'halves' (naive implementation):

```
split :: [a] -> ([a], [a])

split xs =
  let
    l = length xs `div` 2
  in
    (take l xs, drop l xs)
```

Merge Sort Implementation: Combine

Merging two sorted lists into one sorted list:

Merge Sort Implementation: Combine

Merging two sorted lists into one sorted list:

```
merge :: [Integer] -> [Integer] -> [Integer]

merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | y < x       = y : merge (x:xs) ys
  | otherwise = x : merge xs (y:ys)
```

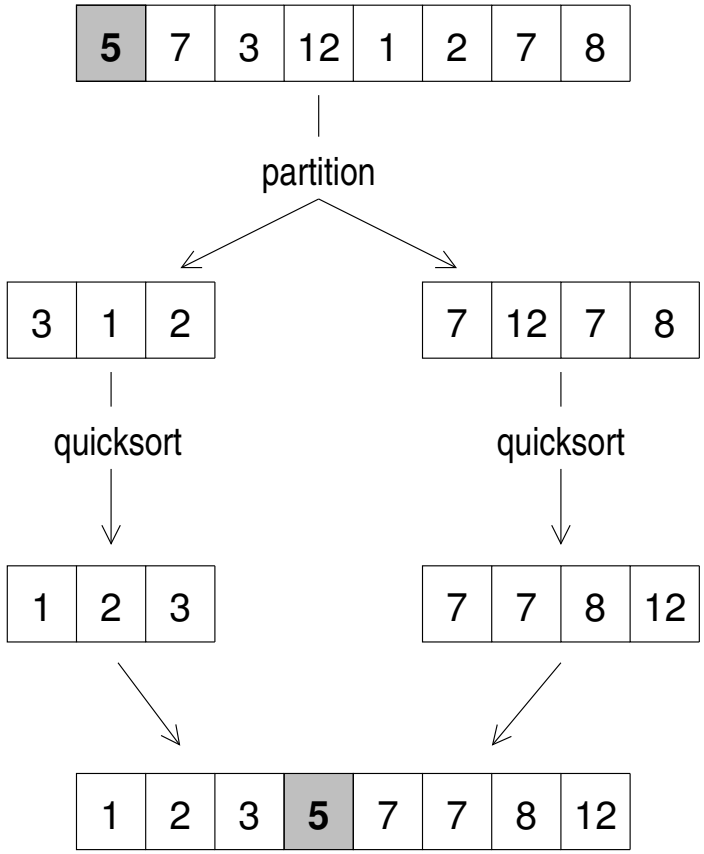
Merge Sort Implementation

Merge Sort Implementation

```
mergeSort :: [Integer] -> [Integer]

mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs =
  let
    (xs1, xs2) = split xs
  in
    merge (mergeSort xs1) (mergeSort xs2)
```

Quicksort (Sir C.A.R. Hoare, 1960)



Quicksort Implementation: Divide

`p` is called the **pivot**. The argument list is partitioned into two lists: one that contains all elements $< \texttt{p}$, and one that contains all elements $\geq \texttt{p}$.

Quicksort Implementation: Divide

`p` is called the **pivot**. The argument list is partitioned into two lists: one that contains all elements $<$ `p`, and one that contains all elements $\geq$ `p`.

```
partition :: Integer -> [Integer] ->
  ([Integer], [Integer])

partition _ [] = ([], [])
partition p (x:xs) =
  let
    (lows, highs) = partition p xs
  in
    if x < p
      then (x:lows, highs)
      else (lows, x:highs)
```

Quicksort Implementation

Quicksort Implementation

```
quicksort :: [Integer] -> [Integer]

quicksort [] = []
quicksort (x:xs) =
  let
    (lows, highs) = partition x xs
  in
    quicksort lows ++ x : quicksort highs
```