## Sorting

Sorting problem: Given a list $A$ with $n$ integers. Rearrange them into non-decreasing order.

The sorting problem is perhaps the most fundamental problem in algorithms.

Instead of integers we can thing about any kind of element that can be compared (`Double`, `String`). In other words, we require a total order on the elements.

There are many direct applications of sorting (catalogs, reports, file listings, etc.)



## Sorting as a tool

There are also many indirect applications of sorting. For instance, algorithms can often be made faster by first sorting the data.

```
def hasDuplicates(L: List[Int]): Boolean = {
  var p = L.sorted
  while (p != Nil && p.tail != Nil) {
    if (p.head == p.tail.head)
      return true
    p = p.tail
  }
  false
}
```

Iterative version



## Insertion Sort

Inserting an element into a sorted list is easy.

```
def insert(L: List[Int], x: Int): List[Int] = {
  L match {
    case Nil => List(x)
    case y::ys => if (x < y) x :: L
                  else y :: insert(ys, x)
  }
}

def insertionSort(L:List[Int]): List[Int] = {
  L match {
    case Nil => Nil
    case x :: xs => insert(insertionSort(xs), x)
  }
}
```



## Selection Sort

Find the minimum from the list, recursively sort the rest.

```
def select(L: List[Int]): (Int, List[Int]) = {
  L match {
    case List(x) => (x, Nil)
    case x :: xs => val (y, ys) = select(xs)
      if (x < y) (x, xs) else (y, x :: ys)
  }
}
def selectionSort(L:List[Int]): List[Int] = {
  if (L.isEmpty) Nil
  else {
    val (x, xs) = select(L)
    x :: selectionSort(xs)
  }
}
```

## In-place insertion sort

When the data is in an array, we can sort it in-place, meaning that we need no extra memory for the sorting.

```
def insertionSort(A: Array[Int], last: Int) {
  if (last > 0) {
    insertionSort(A, last - 1)
    val x = A(last)
    var i = last
    while (i > 0 && x < A(i-1)) {
      A(i) = A(i-1)
      i = i-1
    }
    A(i) = x
  }
}
```



## Iterative version

We can easily remove the recursion:

```
def insertionSort(A: Array[Int]) {
  for (last <- 1 until A.length ) {
    // A(0..last-1) already sorted
    val x = A(last)
    var i = last
    while (i > 0 && x < A(i-1)) {
      A(i) = A(i-1)
      i = i-1
    }
    A(i) = x
  }
}
```



## Bubble Sort

Like in Insertion Sort, we bring the largest element to the top.

```
def bubbleSort(A: Array[Int]) {
  for (last <- A.length - 1 until 0 by -1) {
    for (j <- 0 until last) {
      if (A(j) > A(j+1)) {
        val t = A(j); A(j) = A(j+1); A(j+1) = t
      }
    }
  }
}
```

Bubble-up phase

If nothing happens during a bubble phase, we are done!



## Bubble sort with early termination

We stop when nothing happens in one phase.

```
def bubbleSort(A: Array[Int]) {
  for (last <- A.length - 1 until 0 by -1) {
    var flipped = false
    for (j <- 0 until last) {
      if (A(j) > A(j+1)) {
        val t = A(j); A(j) = A(j+1); A(j+1) = t
        flipped = true
      }
    }
    if (!flipped)
      return
  }
}
```

## Merge-Sort

Let us try divide and conquer:

1. Split the problem into smaller instances.
2. Recursively solve the subproblems.
3. Combine the solutions to solve the original problem.

```
def mergeSort(L: List[Int]): List[Int] = {
  if (L.length > 1) {
    val m = L.length / 2
    val L1 = mergeSort(L take m)
    val L2 = mergeSort(L drop m)
    merge(L1, L2) // combine solutions
  } else
    L
}
```



## Merging

We are given two sorted lists L1 and L2, and we wish to combine them into one sorted list L.

```
def merge(L1: List[Int], L2: List[Int]): List[Int] =
  val L = new ListBuffer[Int]
  var A = L1;  var B = L2
  while (A.nonEmpty && B.nonEmpty) {
    if (A.head < B.head) {
      L += A.head; A = A.tail
    } else {
      L += B.head; B = B.tail
    }
  }
  L ++= A; L ++= B
  L.toList
}
```



## Merge-Sort — Time Analysis

Merging takes $O(n)$ time.

Let $T(n)$ be the time taken by Merge-Sort for $n$ elements. Then $T(1) = O(1)$ and

$$T(n) = 2T(n/2) + O(n)$$

The solution is $O(n \log n)$.



## Quick-sort

Divide and conquer:

1. Split the problem into smaller instances.
2. Recursively solve the subproblems.
3. Combine the solutions to solve the original problem.

In Merge-Sort, the divide step is trivial, and the combine step is where all the work is done.

In Quick-Sort, the combine step is trivial, and all the work is done in the divide step:

1. If $L$ has less than two elements, return. Otherwise, select a pivot $p$ from $L$. Split $L$ into three lists $S$, $E$, and $G$, where
   - $S$ stores the elements of $L$ smaller than $x$,
   - $E$ stores the elements of $L$ equal to $x$, and
   - $G$ stores the elements of $L$ greater than $x$.
2. Recursively sort $S$ and $G$.
3. Form result by concatenating $S$, $E$, and $G$ in this order.

## Quick-Sort

```
def quickSort(L: List[Int]): List[Int] = {
  if (L.length > 1) {
    val p = L((math.random * L.length).toInt)
    val S = quickSort(L filter (_ < p))
    val E = L filter (_ == p)
    val G = quickSort(L filter (_ > p))
    S ::: E ::: G
  } else
    L
}
```



## Quick-Sort Analysis

The running time depends strongly on the choice of the pivot.

In the worst case, it is $O(n^2)$.

In the best case, it is $O(n \log n)$.

If the pivot is selected randomly, the expected running time is $O(n \log n)$.

Quick-Sort can be implemented in-place (using one array only).