ID1020: Mergesort

ch 2.2

Slides adapted from Algoritms 4th Edition, Sedgewick.

Two classic algorithms: mergesort och quicksort

- Key components of the IT-infrastructure of the world.
  - Scientific understanding of their properties has enabled their usage in and enabled the deveolpment of many of the systems we depend on.
  - Quicksort has been selected as one of the top ten most important algorithms of the 20th century.

- Mergesort.

...

- Quicksort.

...

Divide-and-Conquer algorithms

- The principle of Divide-and-Conquer (dela-och-härska) of algorithms:
  - **Divide**: partition the problem in smaller, indepenent, sub-problem. The sub-problems are of the same type as the original problem. The partitioning continues until the original problem has been partitioned into trivial (simple) sub-problems.
  - **Conquer**: solve the sub-problems recursively (or directly).
  - The results are merged into a solution to the original problem.

- Time complexity can normally be solved by analyzing a recurrence relation.

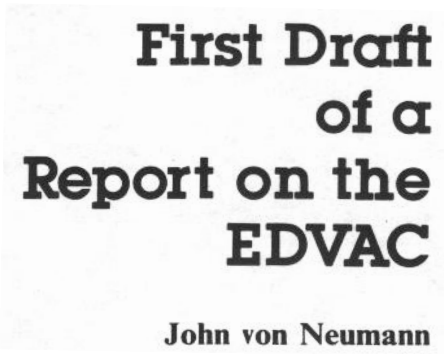
Mergesort

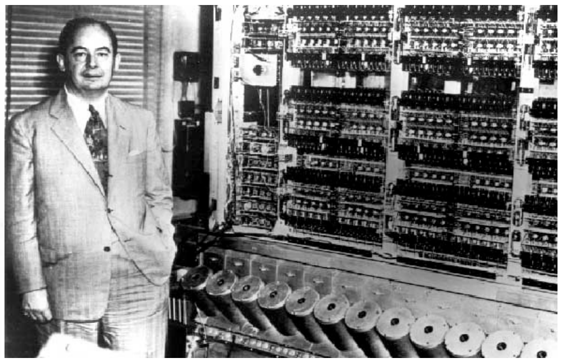
Mergesort

- Basic method.
  - Partition the array in two halves.
  - Recursively sort each half.
  - Merge the halfs.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview





Abstract in-place merge demo

- **Target.** Given two sorted sub-arrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, merge into a sorted array `a[lo]` to `a[hi]`.

	lo				mid	mid+1				hi
a[]	E	E	G	M	R	A	C	E	R	T
	sorted					sorted				

Abstract in-place merge demo

- **Target.** Given two sorted sub-arrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, merge into a sorted array `a[lo]` to `a[hi]`.

	lo									hi
a[]	A	C	E	E	E	G	M	R	R	T

sorterad

Merging

- How do we combine two sorted sub-arrays into a single sorted array?
- Use an auxiliary array.

		a[]												aux[]									
	k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
input		E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-
copy		E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
												0	5										
	0	A										0	6	E	E	G	M	R	A	C	E	R	T
	1	A	C									0	7	E	E	G	M	R		C	E	R	T
	2	A	C	E								1	7	E	E	G	M	R			E	R	T
	3	A	C	E	E							2	7		E	G	M	R			E	R	T
	4	A	C	E	E	E						2	8			G	M	R			E	R	T
	5	A	C	E	E	E	G					3	8			G	M	R				R	T
	6	A	C	E	E	E	G	M				4	8				M	R				R	T
	7	A	C	E	E	E	G	M	R			5	8					R				R	T
	8	A	C	E	E	E	G	M	R	R		5	9									R	T
	9	A	C	E	E	E	G	M	R	R	T	6	10										T
merged result		A	C	E	E	E	G	M	R	R	T												

Abstract in-place merge trace

Merging: Java implementation

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{

   for (int k = lo; k <= hi; k++) {
      aux[k] = a[k];
   }
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

copy

merge

	lo			i	mid			j		hi
aux[]	A	G	L	O	R	H	I	M	S	T

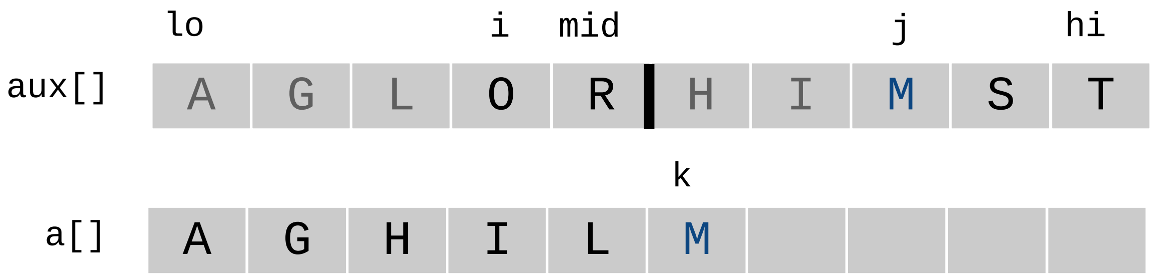
						k				
a[]	A	G	H	I	L	M				

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid,
  int hi) {
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```

copy

merge



Assertions

- Assertion. A statement to check assumptions on the algorithm.
  - Assertions helps identifying bugs in the logic (algorithm).
  - Assertions helpful to document the source code.

- Java assert statement. Throws an exception if the condition is false.

```
assert isSorted(a, lo, hi);
```

- Can be enabled or disabled (i.e. compiled or not compiled).
  ⇒ They do not affect the performance/cost of production code.

```
% java -ea MyProgram   // enable assertions
% java -da MyProgram   // disable assertions
(default)
```

- Best effort. Use assertions to check *invariants*, *pre-conditions* and *post-conditons*, for eg. for an API or method.
  Assume assertions are disabled in production code.

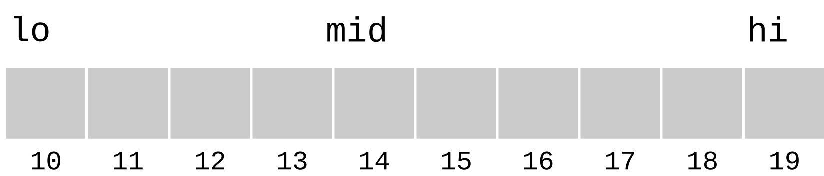
Do not use external arguments to check assertions

- Unit tests checks the *external behaviour of an* API or a method.

Mergesort: Java implementation

```
public class Merge {
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int
hi)    {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a) {
      Comparable[] aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

Why should you not create the auxiliary array here?



Mergesort: trace

lo hi a[]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for top-down mergesort

Mergesort

- https://en.wikipedia.org/wiki/Merge_sort

Mergesort: empiric analysis

- Estimated execution times:
  - A laptop executes 10^8 comparisons/s.
  - A super computer executes 10^{12} comparisons/s.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

- Conclusion. Good algorithms are better than supe computers.

Mergesort: number of comparisons

- **Proposition.** Mergesort compares $\leq N \lg N$ to sort an array of size N.

- **Proof.** The number of comparisons $C(N)$ and array accesses $A(N)$ to sort an array of size N by mergesort satisfies the recurrence relation:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N \quad \text{where } N > 1, \text{ and } C(1) = 0.$$

Left half Right half merge

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \quad \text{where } N > 1, \text{ and } A(1) = 0.$$

- Solve the recurrence relation when N is 2^M:

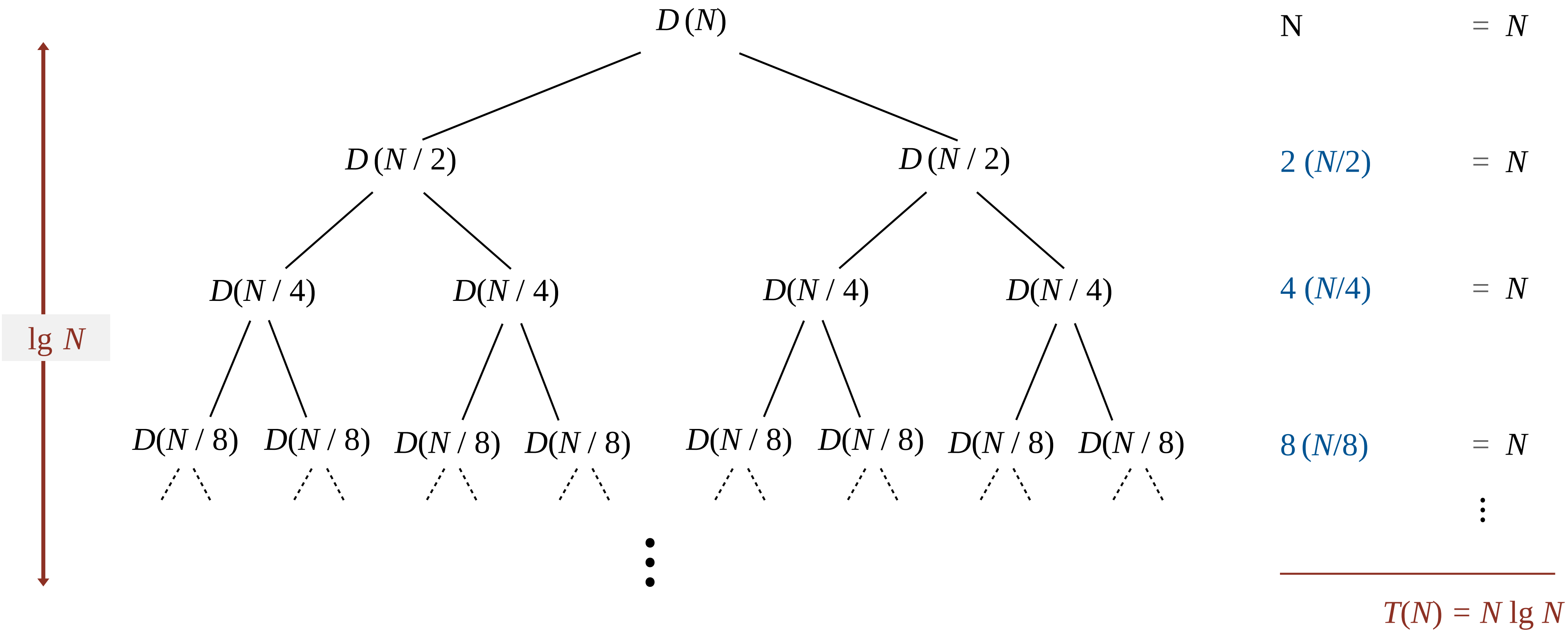
$$D(N) = 2\,D(N/2) + N, \text{ för } N > 1, \text{ and } D(1) = 0.$$

The result can be proven valid for all N
(this assumptions simplifies the analysis)

Divide-och-conquer recurrence: proof by example

- Proposition. IF $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, and $D(1) = 0$,
  THEN $D(N) = N \lg N$.

- Proof 1. [assuming N is 2^M]



Divide-och-conquer recurrence : proof by expansion

- Proposition. IF $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, and $D(1) = 0$,
  THEN $D(N) = N \lg N$.

- Proof 2. [assuming N is 2^M]

$D(N) = 2\,D(N/2) + N$	given
$D(N)\,/\,N = 2\,D(N/2)\,/\,N + 1$	divide both sides by N
$= D(N/2)\,/\,(N/2) + 1$	algebra
$= D(N/4)\,/\,(N/4) + 1 + 1$	apply to first term
$= D(N/8)\,/\,(N/8) + 1 + 1 + 1$	apply to first term again
$\cdots$	
$= D(N/N)\,/\,(N/N) + 1 + 1 + \ldots + 1$	stop applying, D(1) = 0
$= \lg N$	

Divide-och-conquer recurrence: proof by induction

- Proposition. IF $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, and $D(1) = 0$,
  THEN $D(N) = N \lg N$.

- Proof 1. [assuming N is 2^M]

  - Base case: $N = 1$.
  - Induction step: $D(N) = N \lg N$.
  - Target: proove that $D(2N) = (2N) \lg (2N)$.

$D(2N) = 2\,D(N) + 2N$	given
$= 2\,N \lg N + 2N$	inductive hypothesis
$= 2\,N (\lg (2N) - 1) + 2N$	algebra
$= 2\,N \lg (2N)$	QED

Mergesort: number of array accesses

- **Proposition.** Mergesort accesses the array $\leq 6N \lg N$ times to sort an array of size N.

- **Proof.** Number of array accesses $A(N)$ satisfies the recurrence-relation:

$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \;\; \textit{for} \;\; N > 1, \text{ and } A(1) = 0$$

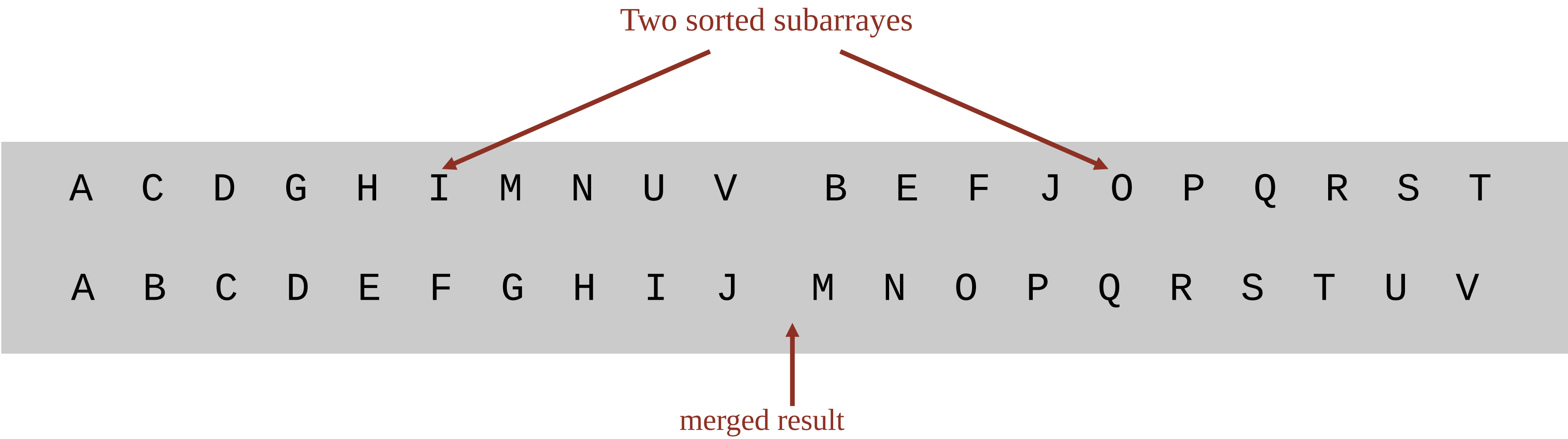
- **Important observation.** Any algorithm with the following structure executes in $N \log N$ time:

```
public static void linearithmic(int N) {
    if (N == 0) {
        return;
    }
    linearithmic(N/2);        Solve two sub-problems
    linearithmic(N/2);        of half the size
    linear(N);                Linear amount of
}                             work
```

- **Known exceptions.** FFT, m.m., ...

Mergesort analys: minneskomplexitet

- Sats. Mergesort använder extra minne proportionellt mot N.
- Bevis. The array `aux[]` needs to be of length N for the last merge.



- Def. An algorithm is ”in-place” if it uses $O(1)$ (often $\leq c \log N$) auxiliary memory.
- Eg. Insertion sort, selection sort, shellsort.

- Challenge 1 (not too hard). Use an `aux[]` array of size ~ ½ N instead of N.
- Chalenge 2 (really hard). *In-place merge*. [Kronrod 1969]

Mergesort: improvements

- Use insertion sort for small sub-arrays.
  - Mergesort (most recursive sorts) has too large overhead for very small sub-arrayer.
  - *Cutoff* ~10 elements – use insertion sort.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi){
   if (hi <= lo + CUTOFF - 1)   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

Mergesort: improvements

- Stop if already sorted.
  - If largest element (key) in first half ≤ smallest element (key) in the second half?
  - Improves performance for partially sorted arrays.

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

Mergesort: Improvements

- Eliminate copying to the auxiliary array. Save time (but not memory) by switching the input and the auxiliary array for each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int
mid, int hi)
{
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++) {
      if      (i > mid)             aux[k] = a[j++];
      else if (j > hi)              aux[k] = a[i++];
      else if (less(a[j], a[i])) aux[k] = a[j++];
      else                          aux[k] = a[i++];
   }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (aux, a, lo, mid);
   sort (aux, a, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

merge from a[] to aux[]

assumes aux[] is initialized to a[] once, before recursive calls

switch roles of aux[] och a[]

Java 6 systemsort

- Base algorithm for sorting of objects = mergesort.
  - Cutoff to insertion sort = 7 (for Strings).
  - Stop if already sorted.
  - Eliminate copying to the auxiliary array trick implemented.

Arrays.sort(a)

http://www.java2s.com/Open-Source/Java/6.0-JDK-Modules/j2me/java/util/Arrays.java.html

Bottom-up Mergesort

Bottom-up mergesort

- Basic method.
  - ”Pass-through” array, merges sub-arrays of size 1.
  - Repeat (iterate) for sub-arrays of size 2, 4, 8, ...

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, aux, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom-up mergesort: Java implementation

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Conclusion. Simple non-recursive version of mergesort.
(uses less execution stack space but is ~10% slower than the recursive version due to worse cache behaviour!)

Time complexity of mergesort

Time complexity for sorting

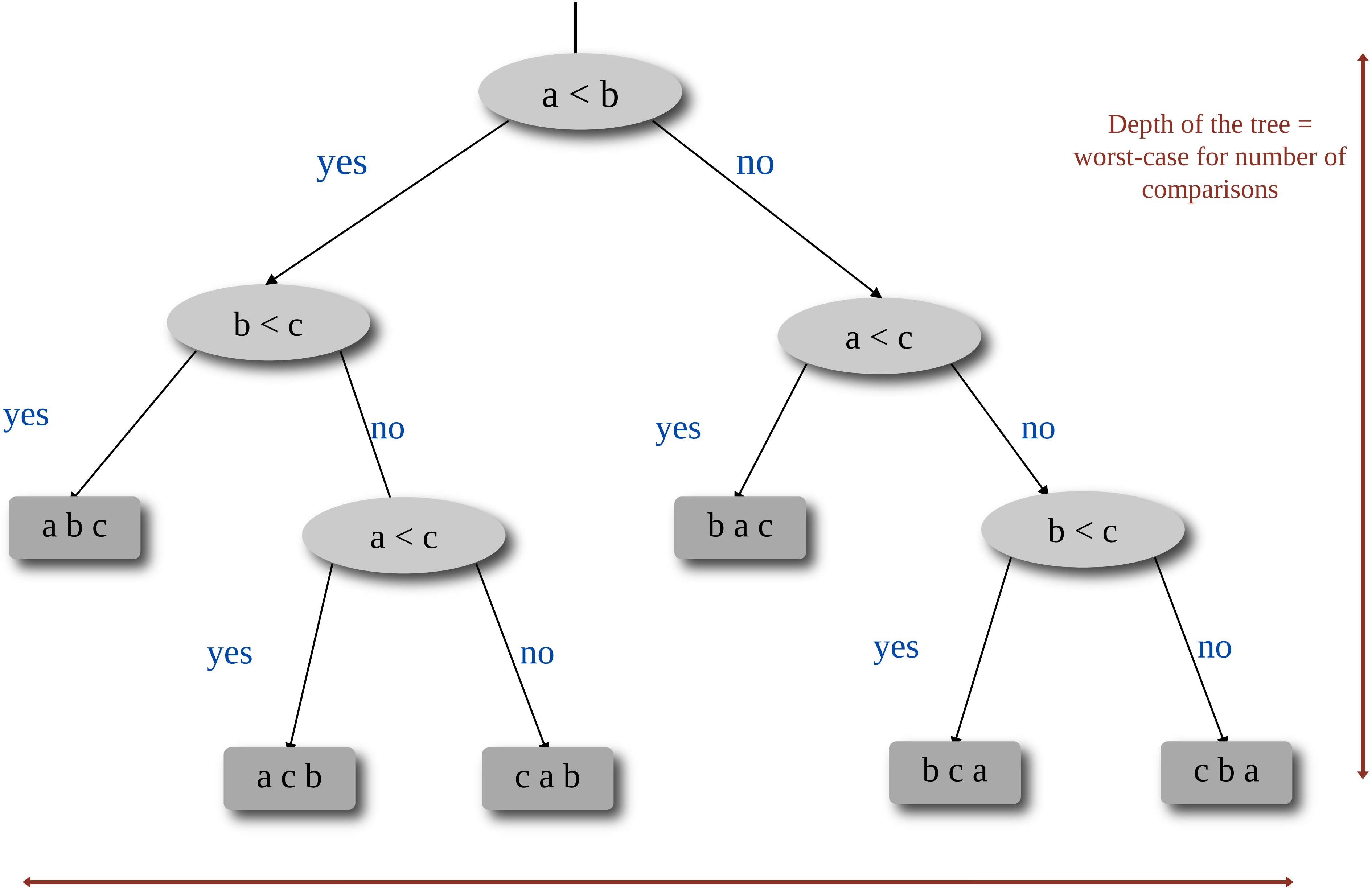
- Cost model. Number of operations.
- Upper bound. Cost guarantee for some algorithm for X.
- Lower bound. No algorithm for X has a cost lower than the bound.
- Optimal algoritm. Algorithm with the best possible cost guarantee for X.

Lower bound ~ upper bound

- Exemple: sorting.
  - Computation model: decision tree.
  - Cost model: # comparisions.
  - Upper bound: $\sim N \lg N$ from mergesort.
  - Lower bound : ?
  - Optimal algorithm: ?

Sort only by comparisions
(t.ex., Java Comparable framework)

Decision tree (for 3 unique keys a, b, and c)

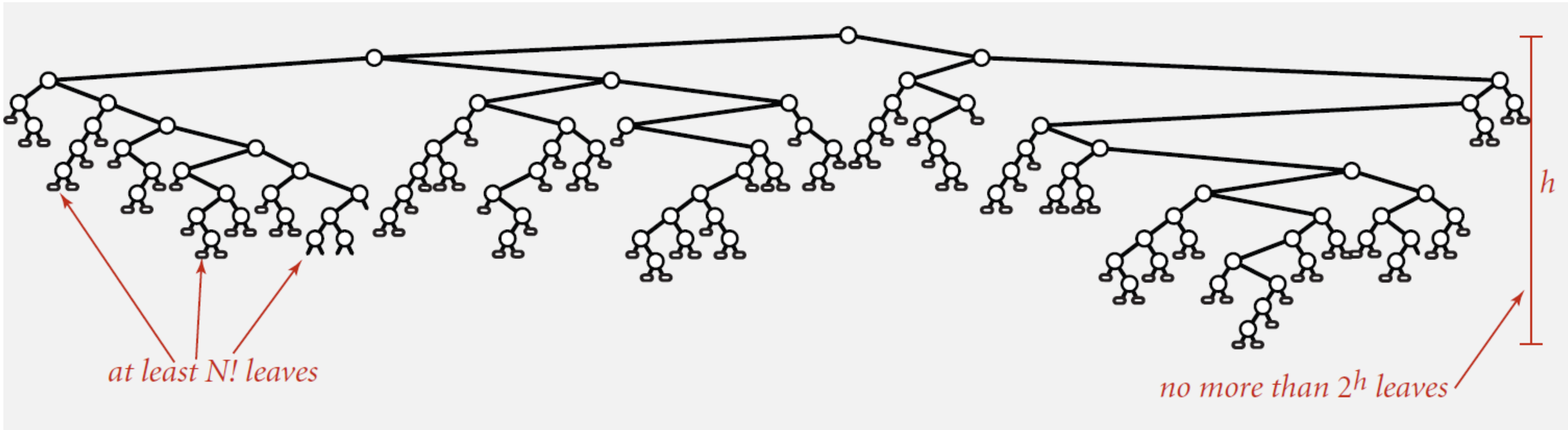


Each leaf corresponds to one (and only one) permutation,
one leaf per possible permutation.

Lower bound for comparison based sorting

- Proposition. No comparson based sorting algorithm can guarantee to sort N unique keys with fewer than $\lg(N!) \sim N \lg N$ comparisons.

- Proof.
  - Assume an array of N unique key values a_1 to a_N.
  - Worst case is limited by the depth (height) h of the decision tree.
  - A binary tree of depth h has at most 2^h leafs.
  - $N!$ Different permutations $\Rightarrow$ at least $N!$ leafs.



Lower bound for comparison based sorting

- Proposition. No comparson based sorting algorithm can guarantee to sort N unique keys with fewer than $\lg(N!) \sim N \lg N$ comparisons.

- Proof.
  - Assume an array of N unique key values a_1 to a_N.
  - Worst case is limited by the depth (height) h of the decision tree.
  - A binary tree of depth h has at most 2^h leafs.
  - $N!$ Different permutations $\Rightarrow$ at least $N!$ leafs.

$$2^h \geq \#\,\text{leafs} \geq N!$$
$$\Rightarrow \quad h \geq \lg(N!) \sim N \lg N$$

Stirling's approximation

Time complexity for comparison based sorting

- Cost model. Number of operations.
- Upper bound. Cost guarantee for some algorithm for X.
- Lower bound. No algorithm for X has a cost lower than the bound.
- Optimal algoritm. Algorithm with the best possible cost guarantee for X.

- Exemple: sorting.
  - Computation model: decision tree.
  - Cost model: # comparisions.
  - Upper bound: ~ $N \lg N$ from mergesort.
  - Lower bound : ~ $N \lg N$.
  - Optimal algorithm = mergesort.

- Primary target for algorithm design: optimal algoritms.

note

- Sorting algorithms that are not based on comparisons may be faster (near O(1))
- Radix, tries
  - Patricia trie used for near O(1) IP route lookups in Linux
- See chapter 5 (which we do not cover in this course)

Complexity results put into context

- Compares? Mergesort is optimal with regard to comparisons.
- Space? Mergesort is not optimal in terms of memory usage (memory complexity).

Lesson. Theory helps us find better solutions.

Eg. Can one design a sorting algorithm which uses at most $\frac{1}{2} N \lg N$ comparisons?

Eg. Can one design a sorting algorithm with optimal time and memory complexity?

Complexity results put into context (cont.)

- The lower bound can be reduced if the algorithm can:
  - Exploit knowledge of how the input is ordered.
    Eg.: insertion sort is O(N) for partially ordered arrayer.
  - Exploit the distribution of key values.
    Eg. : 3-way quicksort only need O(N) comparisions if the number of keys is limited. [nästa föreläsningen]
  - Representation of the keys.
    Eg. : radix sort does not perform comparisons of keys — it accesses data by character/number comparisons.

Mergesort comparators

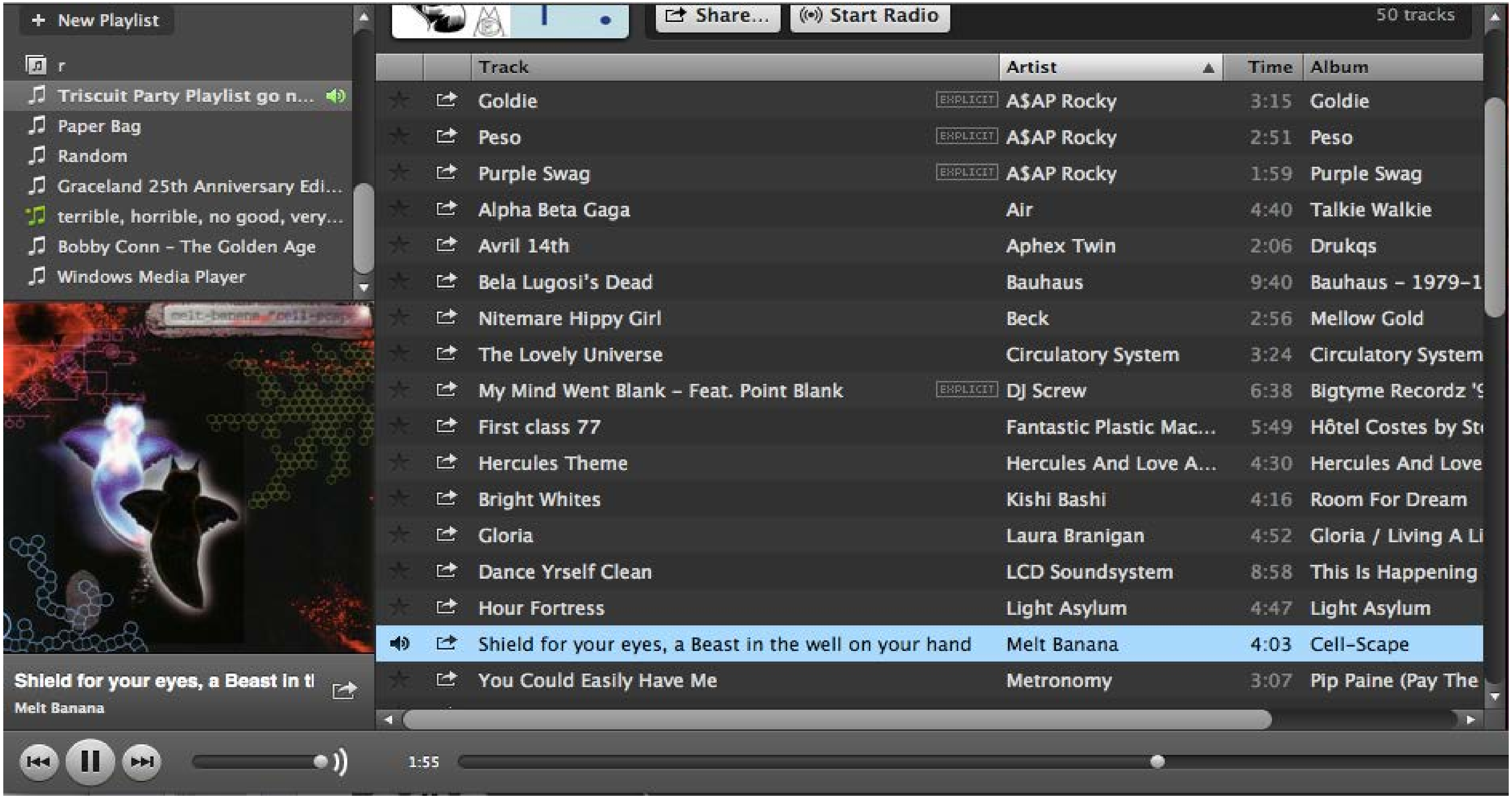
Sort countries by gold medals

NOC	Gold	Silver	Bronze	Total
United States (USA)	46	29	29	104
China (CHN)§	38	28	22	88
Great Britain (GBR)*	29	17	19	65
Russia (RUS)§	24	25	32	81
South Korea (KOR)	13	8	7	28
Germany (GER)	11	19	14	44
France (FRA)	11	11	12	34
Italy (ITA)	8	9	11	28
Hungary (HUN)§	8	4	6	18
Australia (AUS)	7	16	12	35

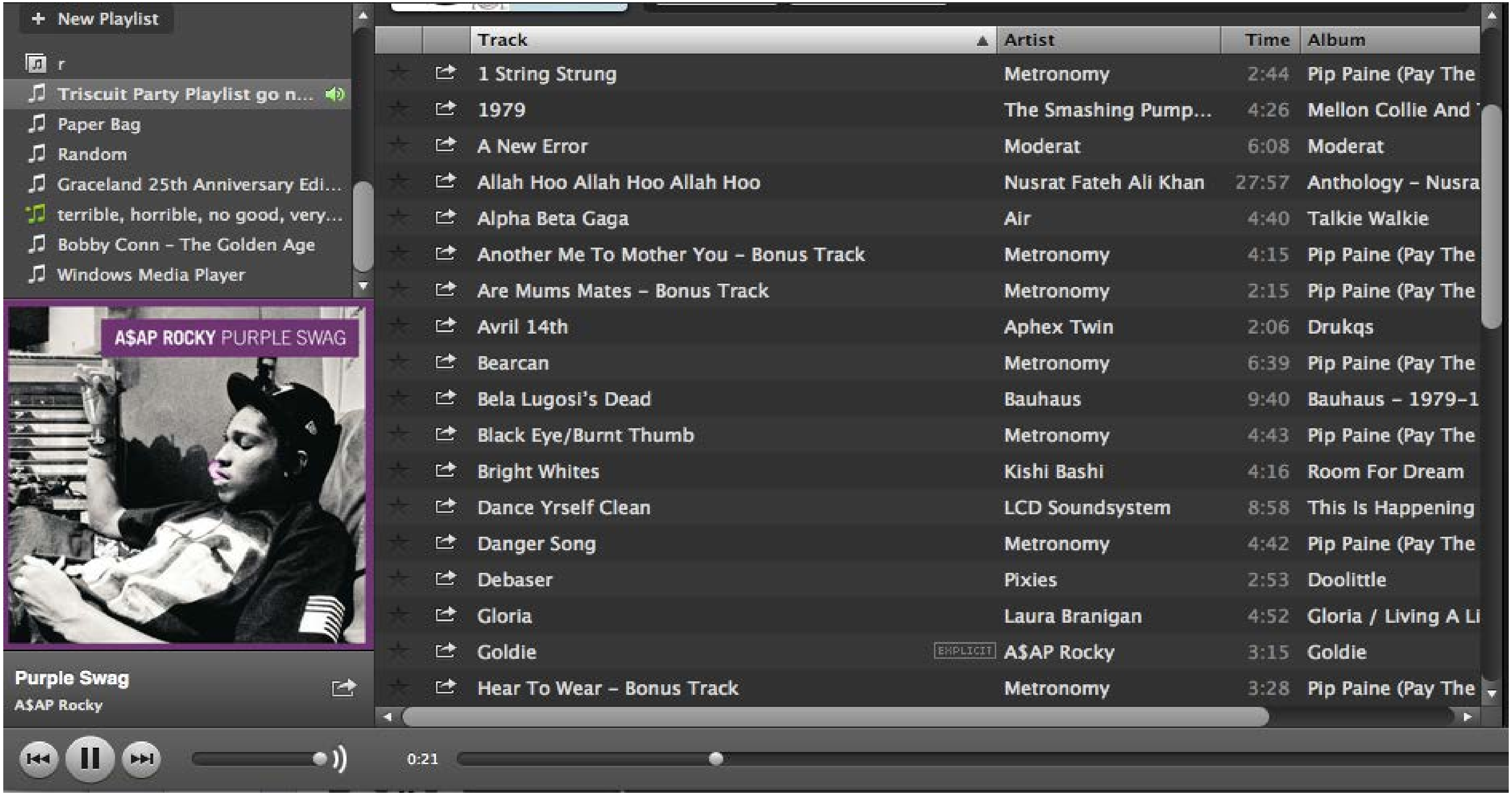
Sort countries by the number of medals

NOC	Gold	Silver	Bronze	Total
United States (USA)	46	29	29	104
China (CHN)§	38	28	22	88
Russia (RUS)§	24	25	32	81
Great Britain (GBR)*	29	17	19	65
Germany (GER)	11	19	14	44
Japan (JPN)	7	14	17	38
Australia (AUS)	7	16	12	35
France (FRA)	11	11	12	34
South Korea (KOR)	13	8	7	28
Italy (ITA)	8	9	11	28

Sort a music library by artist



Sort a music library by tunes



Comparable interface: racap

- Comparable interface: sort according to the natural order of a type.

```
public class Date implements Comparable<Date> {
   private final int month, day, year;

   public Date(int m, int d, int y) {
      month = m;
      day   = d;
      year  = y;
   }
      …
   public int compareTo(Date that)   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

natural order

Comparator interface

- `Comparator` interface: sort according to an alternative order.

public interface Comparator<Key>	
`int compare(Key v, Key w)`	*Compare keys v and w*

- Requirement. Must definie a total order.

natural order	`Now is the time`	
case insensitive	`is Now the time`	pre-1994 order for digraphs ch och ll och rr
Spanish language	`café cafetero cuarto churro nube ñoño`	
British phone book	`McKinley Mackintosh`	

Decouple the comparator interface from the data type

- Use with Java system sort:
  - Create a `Comparator` object.
  - Pass as second argument to `Arrays.sort()`.

uses natural order

Use an alternative order defined by a Comparator<String> object

```
String[] a;
...
Arrays.sort(a);
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
...
Arrays.sort(a, Collator.getInstance(new Locale("es")));
...
Arrays.sort(a, new BritishPhoneBookOrder());
...
```

- Conclusion. Decouple the definition of the data type from the definition of what it means to compare two objects of the same type.

Comparator interface: to use sorting libraries from the book

- To use comparators in the sorting methods implemented in the book:
  - Use `Object` instead of `Comparable`.
  - Pass a `Comparator` to `sort()` and `less()` and use it in `less()`.

insertion sort using a Comparator

```
public static void sort(Object[] a, Comparator comparator)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w) {
   return c.compare(v, w) < 0;
}

private static void exch(Object[] a, int i, int j){
   Object swap = a[i]; a[i] = a[j]; a[j] = swap;
}
```

To implement a Comparator interface

- To implement a comparator:
  - Define a (nested) class which implements the Comparator interface.
  - Implement the compare( ) method.

```
public class Student {
private final String name;
   private final int section;
   ...

   public static class ByName implements Comparator<Student>
   {
     public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }

   public static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

To implement a Comparator interface

- To implement a comparator:
  - Define a (nested) class which implements the `Comparator` interface.
  - Implement the `compare()` method.

```
Arrays.sort(a, new Student.ByName());
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Arrays.sort(a, new Student.BySection());
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

To implement a Comparator interface

- To implement a comparator:
  - Define a (nested) class which implements the Comparator interface.
  - Implement the compare( ) method.

```
public class Student {
   public static final Comparator<Student> BY_NAME = new ByName();
   public static final Comparator<Student> BY_SECTION = new BySection();
   private final String name;
   private final int section;
   ...

   public static class ByName implements Comparator<Student>
   {
     public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }

   public static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

To implement a Comparator interface

- To implement a comparator:
  - Define a (nested) class which implements the `Comparator` interface.
  - Implement the `compare()` method.
  - Give access to static `Comparator` objects.

```
Arrays.sort(a, Student.BY_NAME);
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Arrays.sort(a, Student.BY_SECTION);
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

Mergesort stability

Stability

- Common application. First, sort by name; then sort by "Section".

```
Selection.sort(a, new Student.ByName());
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Selection.sort(a, new Student.BySection());
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

- @#%&@! Students in section 3 are no longer sorted by name.

- A stable sort maintains the relative order of elements with equal keys.

Stability

- Which sorting algorithms are stable?
  We need to check the algorithm (and implementation) to undertsand if it is stable or not.

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

no longer sorted by time

still sorted by time

Stability: insertion sort

- **Proposition.** Insertion sort is stable.

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
   }
}
```

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

Prof. If two elements have equal keys, insertion sort never moves one element before the other, (it never re-orders elements with equal keys)

Stability: selection sort

- Proposition. Selection sort is not stable.

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }
}
```

i	min	0	1	2
0	2	B_1	B_2	A
1	1	A	B_2	B_1
2	2	A	B_2	B_1
		A	B_2	B_1

- Proof by counter example. (long) distance swapping may move an element passed another with equal keys.

Stabilitet: shellsort

- Proposition. Shellsort sort is not stable.

```
public class Shell {
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1;
      while (h >= 1)
      {
         for (int i = h; i < N; i++)
         {
            for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }
         h = h/3;
      }
   }
}
```

h	0	1	2	3	4
	B_1	B_2	B_3	B_4	A_1
4	A_1	B_2	B_3	B_4	B_1
1	A_1	B_2	B_3	B_4	B_1
	A_1	B_2	B_3	B_4	B_1

- Proof by counter example. (long) distance swapping may move an element passed another with equal keys.

Stability: mergesort

- Proposition. Mergesort is stable.

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {  /* as before */  }
}
```

- Proof. It is sufficient to verify that a merge operation is stable.

Stability: mergesort

- Proposition. The merge operation is stab.

```
private static void merge(...)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

0	1	2	3	4	5	6	7	8	9	10
A_1	A_2	A_3	B	D	A_4	A_5	C	E	F	G

- Proof. two equal keys on the same half won't change relative order;
- two equal keys on different halves won't change relative order because we take from left subarray if equal

Sort summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✔		$N \log_3 N$	?	$c N^{3/2}$	tight code; subquadratic
mergesort		✔	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail