Two classic sorting algorithms

Critical components in the world’s computational infrastructure.
- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

**Mergesort.** [this lecture]
- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

**Quicksort.** [next lecture]
- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...

---

**2.2 MERGESORT**

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability

---

**Mergesort overview**

---

**Mergesort**

Basic plan.
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

---

**Example**

<table>
<thead>
<tr>
<th>input</th>
<th>M E R G E S O R T E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>E E G M O R R S</td>
</tr>
<tr>
<td>sort right half</td>
<td>A E E L M P T X</td>
</tr>
<tr>
<td>merge results</td>
<td>A E E E E G L M M O P R R S T X</td>
</tr>
</tbody>
</table>

---

First Draft of a Report on the EDVAC

John von Neumann
Abstract in-place merge demo

Goal. Given two sorted subarrays \(a[lo] \) to \(a[mid] \) and \(a[mid+1] \) to \(a[hi] \), replace with sorted subarray \(a[lo] \) to \(a[hi] \).

<table>
<thead>
<tr>
<th>lo</th>
<th>mid</th>
<th>mid+1</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

sorted \hspace{1cm} \text{sorted}

Merging: Java implementation

```java
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
}
```

Assertions

**Assertion.** Statement to test assumptions about your program.
- Helps detect logic bugs.
- Documents code.

**Java assert statement.** Throws exception unless boolean condition is true.

**Can enable or disable at runtime.** \(\Rightarrow \) No cost in production code.

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

**Best practices.** Use assertions to check internal invariants; assume assertions will be disabled in production code.
Mergesort: Java implementation

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

Mergesort: trace

```
<table>
<thead>
<tr>
<th>lo</th>
<th>mid</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>
```

result after recursive call

Mergesort: animation

http://www.sorting-algorithms.com/merge-sort

50 random items

50 reverse-sorted items

http://www.sorting-algorithms.com/merge-sort
Mergesort: empirical analysis

Running time estimates:
- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

Bottom line. Good algorithms are better than supercomputers.

<table>
<thead>
<tr>
<th>insertion sort (N^2)</th>
<th>mergesort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
</tr>
</tbody>
</table>

Mergesort: number of compares and array accesses

Proposition. Mergesort uses at most $N \lg N$ compares and $6N \lg N$ array accesses to sort any array of size $N$.

Pf sketch. The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size $N$ satisfy the recurrences:

$$C(N) = C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N$$

for $N > 1$, with $C(1) = 0$.

$$A(N) = A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N$$

for $N > 1$, with $A(1) = 0$.

We solve the recurrence when $N$ is a power of 2. \(\rightarrow\) result holds for all $N$

$$D(N) = 2D(N/2) + N$$

for $N > 1$, with $D(1) = 0$.

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming $N$ is a power of 2]

Given:
- Divide both sides by $N$
- Apply first term
- Apply first term again
- Stop applying, $D(1) = 0$

Divide-and-conquer recurrence: proof by expansion

Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 2. [assuming $N$ is a power of 2]

$$D(N) = 2D(N/2) + N$$

$$D(N)/N = 2D(N/2)/N + 1$$

$$= D(N/2)/(N/2) + 1$$

$$= D(N/4)/(N/4) + 1 + 1$$

$$= D(N/8)/(N/8) + 1 + 1 + 1$$

$$\ldots$$

$$= D(N/N)/(N/N) + 1 + 1 + \ldots + 1$$

$$= \lg N$$
Divide-and-conquer recurrence: proof by induction

**Proposition.** If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

**Pf 3.** [assuming $N$ is a power of 2]
- **Base case:** $N = 1$.
- **Inductive hypothesis:** $D(N) = N \lg N$.
- **Goal:** show that $D(2N) = (2N) \lg (2N)$.

\[
D(2N) = 2D(N) + 2N \\
= 2 \cdot N \lg N + 2N \\
= 2N (\lg (2N) - 1) + 2N \\
= 2N \lg (2N)
\]

**QED**

**Mergesort analysis: memory**

**Proposition.** Mergesort uses extra space proportional to $N$.

**Pf.** The array `aux[]` needs to be of size $N$ for the last merge.

**Def.** A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

**Ex.** Insertion sort, selection sort, shellsort.

**Challenge for the bored.** In-place merge. [Kronrod, 1969]

**Mergesort: practical improvements**

Use insertion sort for small subarrays.
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $\sim 7$ items.

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

**Mergesort: practical improvements**

**Stop if already sorted.**
- Is biggest item in first half $\leq$ smallest item in second half?
- Helps for partially-ordered arrays.

```java
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: practical improvements

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```java
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[i], a[j])) aux[k] = a[i++];
        else aux[k] = a[j++];
}
```

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, hi);
    merge(aux, a, lo, mid, hi);
}

switch roles of aux[] and a[]

2.2 Mergesort

- Mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability

Bottom-up mergesort

Basic plan.
- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ...
Bottom-up mergesort: Java implementation

```java
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));
    }
}
```

Bottom line. Simple and non-recursive version of mergesort.

but about 10% slower than recursive, top-down mergesort on typical systems

Bottom-up mergesort: visual trace

2
4
8
16
32

Visual trace of bottom-up mergesort

Complexity of sorting

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem $X$.

Model of computation. Allowable operations.

Cost model. Operation count(s).

Upper bound. Cost guarantee provided by some algorithm for $X$.

Lower bound. Proven limit on cost guarantee of all algorithms for $X$.

Optimal algorithm. Algorithm with best possible cost guarantee for $X$.

Example: sorting.

- Model of computation: decision tree.
- Cost model: # compares.
- Upper bound: $\sim N \lg N$ from mergesort.
- Lower bound: $?$
- Optimal algorithm: $?$
Proposition. Any compare-based sorting algorithm must use at least \( \lg (N!) \sim N \lg N \) compares in the worst-case.

\textbf{Pf.} 
- Assume array consists of \( N \) distinct values \( a_1 \) through \( a_N \).
- Worst case dictated by height \( h \) of decision tree.
- Binary tree of height \( h \) has at most \( 2^h \) leaves.
- \( N! \) different orderings \( \Rightarrow \) at least \( N! \) leaves.

\[ 2^h \geq \# \text{leaves} \geq N! \]
\[ \Rightarrow h \geq \lg (N!) \sim N \lg N \]

Stirling's formula

Complexity of sorting

Model of computation. Allowable operations.
Cost model. Operation count(s).
Upper bound. Cost guarantee provided by some algorithm for \( X \).
Lower bound. Proven limit on cost guarantee of all algorithms for \( X \).
Optimal algorithm. Algorithm with best possible cost guarantee for \( X \).

Example: sorting.
- Model of computation: decision tree.
- Cost model: \# compares.
- Upper bound: \( \sim N \lg N \) from mergesort.
- Lower bound: \( \sim N \lg N \).
- Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.
Complexity results in context

**Compares?** Mergesort is optimal with respect to number compares.

**Space?** Mergesort is not optimal with respect to space usage.

Lessons. Use theory as a guide.

**Ex.** Design sorting algorithm that guarantees \( \frac{1}{2} N \lg N \) compares?

**Ex.** Design sorting algorithm that is both time- and space-optimal?

### 2.2 MERGESORT

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability

Sort music library by artist name
**Comparable interface: sort using a type's natural order.**

```java
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;
    }
}
```

**Comparator interface: system sort**

To use with Java system sort:
- Create Comparator object.
- Pass as second argument to Arrays.sort().

```java
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());
```

**Bottom line.** Decouples the definition of the data type from the definition of what it means to compare two objects of that type.
Comparator interface: using with our sorting libraries

To support comparators in our sort implementations:
- Use Object instead of Comparable.
- Pass Comparator to sort() and less() and use in less().

insertion sort using a Comparator

```java
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);
}
```

Comparator interface: implementing

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

```java
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;
    ...
    one Comparator for the class
    private static class ByName implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        { return v.name.compareTo(w.name); }
    }
    private static class BySection implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        { return v.section - w.section; }
    }
}
```

this technique works here since no danger of overflow

Comparator interface: implementing

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

```java
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; }
```

Polar order

**Polar order.** Given a point $p$, order points by polar angle they make with $p$.

**Application.** Graham scan algorithm for convex hull. [see previous lecture]

**High-school trig solution.** Compute polar angle $\theta$ w.r.t. $p$ using $\text{atan2}()$.

**Drawback.** Evaluating a trigonometric function is expensive.
2.2 MERGESORT

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability

Comparator interface: polar order

```java
public class Point2D
{
    public final Comparator<Point2D> POLAR_ORDER = new PolarOrder();
    private final double x, y;
    ...

    private static int ccw(Point2D a, Point2D b, Point2D c)
    {
        return a.compareTo(b) - a.compareTo(c);
    }

    public class PolarOrder implements Comparator<Point2D>
    {
        public int compare(Point2D q1, Point2D q2)
        {
            double dy1 = q1.y - y;
            double dy2 = q2.y - y;
            if (dy1 == 0 && dy2 == 0) { ... }
            else if (dy1 > 0 && dy2 < 0) return -1;
            else if (dy2 >= 0 && dy1 < 0) return +1;
            else return -ccw(Point2D.this, q1, q2);
        }
    }
}
```

Stability

A typical application. First, sort by name; then sort by section.

Selection.sort(a, Student.BY_NAME);
Selection.sort(a, Student.BY_SECTION);

<table>
<thead>
<tr>
<th>Name</th>
<th>Grade</th>
<th>ID</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A 664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C 874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A 991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A 884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A 766-093-8783</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B 766-093-8783</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B 898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A 232-343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A 664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A 232-343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A 991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A 884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A 664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B 898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B 766-093-8783</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C 874-088-1212</td>
<td>121 Whitman</td>
</tr>
</tbody>
</table>

@##@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.
**Stability**

**Q.** Which sorts are stable?

**A.** Insertion sort and mergesort (but not selection sort or shellsort).

<table>
<thead>
<tr>
<th>Time</th>
<th>Location (Not Stable)</th>
<th>Location (Stable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago 09:00:00</td>
<td>Chicago 09:25:52</td>
<td>Chicago 09:00:00</td>
</tr>
<tr>
<td>Phoenix 09:00:03</td>
<td>Los Angeles 09:03:13</td>
<td>Phoenix 09:00:59</td>
</tr>
<tr>
<td>Houston 09:00:13</td>
<td>Dallas 09:19:46</td>
<td>Houston 09:01:10</td>
</tr>
<tr>
<td>Chicago 09:00:59</td>
<td>Chicago 09:21:05</td>
<td>Chicago 09:19:32</td>
</tr>
<tr>
<td>Houston 09:01:10</td>
<td>Chicago 09:19:46</td>
<td>Houston 09:01:10</td>
</tr>
<tr>
<td>Chicago 09:03:13</td>
<td>Chicago 09:21:05</td>
<td>Chicago 09:19:32</td>
</tr>
<tr>
<td>Seattle 09:10:11</td>
<td>Chicago 09:35:21</td>
<td>Chicago 09:25:52</td>
</tr>
<tr>
<td>Seattle 09:10:25</td>
<td>Chicago 09:35:21</td>
<td>Chicago 09:35:21</td>
</tr>
<tr>
<td>Phoenix 09:14:25</td>
<td>Houston 09:01:10</td>
<td>Houston 09:01:10</td>
</tr>
<tr>
<td>Chicago 09:19:32</td>
<td>Houston 09:00:13</td>
<td>Houston 09:00:13</td>
</tr>
<tr>
<td>Chicago 09:19:46</td>
<td>Phoenix 09:37:44</td>
<td>Phoenix 09:00:03</td>
</tr>
<tr>
<td>Chicago 09:21:05</td>
<td>Phoenix 09:00:03</td>
<td>Phoenix 09:14:25</td>
</tr>
<tr>
<td>Seattle 09:22:54</td>
<td>Seattle 09:10:25</td>
<td>Seattle 09:10:25</td>
</tr>
<tr>
<td>Chicago 09:25:52</td>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:25</td>
</tr>
<tr>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:11</td>
<td>Seattle 09:22:54</td>
</tr>
<tr>
<td>Phoenix 09:37:44</td>
<td>Seattle 09:22:54</td>
<td>Seattle 09:36:14</td>
</tr>
</tbody>
</table>

**Note.** Need to carefully check code ("less than" vs. "less than or equal to").

**Stability: selection sort**

**Proposition.** Selection sort is not stable.

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

**Pf by counterexample.** Long-distance exchange might move an item past some equal item.

**Stability: shellsort**

**Proposition.** Shellsort sort is not stable.

```java
public class Shellsort
{
    public static Shellsort 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++)
                if (j < h) h = less(a[j], a[h]);
                exch(a, j, h);
            h = h/3;
        }
    }
}
```

**Pf by counterexample.** Long-distance exchanges.
Stability: mergesort

**Proposition.** Mergesort is stable.

```java
public class Merge
{
    private static Comparable[] aux;
    private static void merge(Comparable[] a, int lo, int mid, int hi)
    {
        /* as before */
    }

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

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

**Pf.** Suffices to verify that merge operation is stable.

Stability: mergesort

**Proposition.** Merge operation is stable.

```java
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++];
    }
}
```

**Pf.** Takes from left subarray if equal keys.