Mergesort

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

Mergesort example

input: E E G M R R S T

sort left half: E E G M R

sort right half: A E E L M P T X

merge results: A E E E E G L M O P R R S T X

Mergesort overview

Abstract in-place merge

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>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>R</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sorted

sorted
Abstract in-place merge

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

1. \( a[] \)  
   \[E E G M R A C E R T\]

   **copy to auxiliary array**

   \( aux[] \)  
   
   

2. \( aux[] \)  
   
   

   **compare minimum in each subarray**

   \( aux[] \)  
   \[E E G M R A C E R T\]

   \( k \)

   \( i \)  
   \( j \)
Abstract in-place merge

Goal. Given two sorted subarrays $a_{[lo]}$ to $a_{[mid]}$ and $a_{[mid+1]}$ to $a_{[hi]}$, replace with sorted subarray $a_{[lo]}$ to $a_{[hi]}$.

```
a[ lo] A E G M R A C E R T
    k
```

compare minimum in each subarray

```
aux[] E E G M R A C E R T
  i
  j
```

Abstract in-place merge

```
a[ lo] A C G M R A C E R T
    k
```

compare minimum in each subarray

```
aux[] E E G M R A C E R T
  i
  j
```

Abstract in-place merge

```
a[ lo] A C G M R A C E R T
    k
```

compare minimum in each subarray

```
aux[] E E G M R A C E R T
  i
  j
```

Abstract in-place merge

```
a[ lo] A C G M R A C E R T
    k
```

compare minimum in each subarray

```
aux[] E E G M R A C E R T
  i
  j
```
Abstract in-place merge

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

\[
\begin{array}{c}
13
\end{array}
\]

Abstract in-place merge

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

\[
\begin{array}{c}
14
\end{array}
\]

Abstract in-place merge

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

\[
\begin{array}{c}
15
\end{array}
\]

Abstract in-place merge

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

\[
\begin{array}{c}
16
\end{array}
\]
Goal. Given two sorted subarrays $a_{lo}$ to $a_{mid}$ and $a_{mid+1}$ to $a_{hi}$, replace with sorted subarray $a_{lo}$ to $a_{hi}$.

```
\text{aux[]}\
\begin{array}{cccccccc}
E & E & G & M & R & A & C & E & R & T
\end{array}
```

compare minimum in each subarray

```
\text{aux[]}\
\begin{array}{cccccccc}
E & E & G & M & R & A & C & E & R & T
\end{array}
```

```python

Abstract in-place merge

Goal. Given two sorted subarrays $a_{lo}$ to $a_{mid}$ and $a_{mid+1}$ to $a_{hi}$, replace with sorted subarray $a_{lo}$ to $a_{hi}$.

```
\text{aux[]}\
\begin{array}{cccccccc}
E & E & G & M & R & A & C & E & R & T
\end{array}
```

compare minimum in each subarray

```
\text{aux[]}\
\begin{array}{cccccccc}
E & E & G & M & R & A & C & E & R & T
\end{array}
```
Abstract in-place merge

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

\[
\begin{array}{ccccccccc}
\text{a[lo]} & A & C & E & E & E & G & M & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]

compare minimum in each subarray

\[
\begin{array}{ccccccccc}
\text{a[lo]} & A & C & E & E & E & G & M & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]

one subarray exhausted, take from other

\[
\begin{array}{ccccccccc}
\text{a[lo]} & A & C & E & E & E & G & M & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]
Abstract in-place merge

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

\[ k \]

one subarray exhausted, take from other

\[ aux[] \]

both subarrays exhausted, done

\[ aux[] \]
Mergesort: Java implementation

```java
public class Merge
{
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
    {
        int i = lo, j = mid + 1;
        aux[k++] = a[i++];
        while (i <= mid && j <= hi)
        {
            if (less(aux[j], aux[i]))
                aux[k++] = aux[j++];
            else if (j > hi)
                aux[k++] = aux[i++];
            else if (i > mid)
                aux[k++] = aux[j++];
        }
        for (int k = lo; k <= hi; k++)
            a[k] = aux[k];
    }
    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>i</th>
<th>mid</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>G</td>
<td>S</td>
<td>W</td>
<td>M</td>
<td>H</td>
<td>V</td>
<td>N</td>
<td>I</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>S</td>
<td>E</td>
<td>X</td>
</tr>
</tbody>
</table>
```
Mergesort: empirical analysis

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

<table>
<thead>
<tr>
<th></th>
<th>insertion sort (N)</th>
<th>mergesort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>computer</td>
<td>thousand</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of compares and array accesses

Proposition. Mergesort uses at most $N \log N$ compares and $6N \log N$ 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:

$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N$ for $N > 1$, with $C(1) = 0$.

$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N$ for $N > 1$, with $A(1) = 0$.

We solve the recurrence when $N$ is a power of 2.

$D(N) = 2D(\lfloor N/2 \rfloor) + N$, for $N > 1$, with $D(1) = 0$. 
**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[i++];
        else if (j > hi) a[k] = aux[j++];
        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
}
```

**Proof:** Each merge uses at most $6N$ array accesses (2N for the copy, 2N for the move back, and at most 2N for compares). The result follows from the same argument as for Proposition F.

---

**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 $D(N) = N\log N$.

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

```
given
D(N) = 2D(N/2) + N
D(N)/N = 2D(N/2)/N + 1
      = 2(D(N/4)/N/4) + 1 + 1
      = 2(D(N/8)/N/8) + 1 + 1 + 1
      . . .
      = D(N/2^k)/N/2^k) + 1 + 1 + ... + 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 $D(N) = N\log N$.

**Pf 3.** [assuming $N$ is a power of 2]

- **Base case:** $N = 1$.
- **Inductive hypothesis:** $D(N) = N\log N$.
- **Goal:** show that $D(2N) = (2N)\log (2N)$.

```
given
D(2N) = 2D(N) + 2N
      = 2N\log N + 2N
      = 2N(\log (2N) − 1) + 2N
      = 2N\log (2N)
```

**QED**
Mergesort analysis: memory

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

**Pf.** The array $\text{aux}[]$ 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 $\approx 7$ 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);
}
```

**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);
}
```

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

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

```
```
Mergesort: visualization

Visual trace of top-down mergesort for with cutoff for small subarrays first subarray second subarray first merge first half sorted second half sorted result

Bottom-up mergesort

Basic plan.
• Pass through array, merging subarrays of size 1.
• Repeat for subarrays of size 2, 4, 8, 16,....

Bottom line.
No recursion needed!

Bottom-up mergesort: Java implementation

public class MergeBU
{
    private static Comparable[] aux;
    private static void merge(Comparable[] a, int lo, int mid, int hi)
    {
        /* as before */
    }
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        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, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
Bottom-up mergesort: visual trace

http://bl.ocks.org/mbostock/39566aaca95eb03ddd526

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: ?

Decision tree (for 3 distinct items $a$, $b$, and $c$)

lower bound -- upper bound

height of tree = worst-case number of

(at least) one leaf for each possible ordering
**Compare-based lower bound for sorting**

**Proposition.** Any compare-based sorting algorithm must use at least \( \lg (N!) \sim N \lg N \) compares in the worst-case.

**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**

**Other operations?** Mergesort is optimal with respect to number of compares (e.g., but not with respect to number of array accesses).

**Space?**
- Mergesort is not optimal with respect to space usage.
- Insertion sort, selection sort, and shellsort are space-optimal.

**Challenge.** Find an algorithm that is both time- and space-optimal. [stay tuned]

**Lessons.** Use theory as a guide.

**Ex.** Don’t try to design sorting algorithm that guarantees \( \frac{1}{2} N \lg N \) compares.
Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about:
- The initial order of the input.
- The distribution of key values.
- The representation of the keys.

**Partially-ordered arrays.** Depending on the initial order of the input, we may not need $N \lg N$ compares.

**Duplicate keys.** Depending on the input distribution of duplicates, we may not need $N \lg N$ compares.

**Digital properties of keys.** We can use digit/character compares instead of key compares for numbers and strings.

Comparator interface

**Comparator interface:** sort using an alternate order.

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

**Required property.** Must be a total order.

**Ex.** Sort strings by:
- Natural order:
  - Now is the time
  - is Now the time
- Case insensitive:
  - café cafetero cuarto churro nube año
  - McAlinley Mackintosh
- Spanish:
  - cafe cafetero cuarto churro nube año
- British phone book:
  - "McKinley Macintosh"
- ...
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 it in less().

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;  }

insertion sort using a Comparator

Comparator interface: implementing

To implement a comparator:

• Define a (nested) class that 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;
    ...
    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;  }
    }
}

Stability

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

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

@#$%&@! 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).

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

Proposition. Insertion sort is stable.

```java
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);
    }
}
```

Pf. Equal items never move past each other.

Proposition. Selection sort is not stable.

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

Proposition. Shellsort is not stable.

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

Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.

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 */
   }
}

Stability: mergesort

Proposition. Merge operation is stable.

Pf. Takes from left subarray if equal keys.

private static void merge(Comparable[] a, 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++];
   }

A 1 2 3 4  5 6 7 8 9 10
A A A B D  A A  C E F G