Mergesort and Quicksort

Two great sorting algorithms.
- Full scientific understanding of their properties has enabled us to hammer them into practical system sorts.
- Occupy a prominent place in world’s computational infrastructure.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.
- Java sort for objects.
- Perl, Python stable.

Quicksort.
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

Mergesort

Basic plan:
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Input:
MERGESORT EXAMPLE

Sort left:
EGMRORS T EXAMP LE

Sort right:
EGMRORS AEELMPT X

Merge:
ABEEEBGLMOPRRSTX

Mergesort: Example
**Merging**

Combine two pre-sorted lists into a sorted whole.

**How to merge efficiently?** Use an auxiliary array.

```java
private static void merge(Comparable[] a, Comparable[] aux, int l, int m, int r)
{
    for (int k = l; k < r; k++)
        aux[k] = a[k];
    int i = l, j = m;
    for (int k = l; k < r; k++)
    {
        a[k] = aux[i] ;
        if  (i >= m)
        else if  (j >= r)
        else if (less(aux[j], aux[i]))
    {

```

**Mergesort: Java implementation of recursive sort**

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

**Mergesort analysis: Memory**

Q. How much memory does mergesort require?
A. Too much!

- Original input array = N.
- Auxiliary array for merging = N.
- Local variables: constant.
- Function call stack: \( \log_2 N \) [stay tuned].
- Total = \( 2N + O(\log N) \).

Q. How much memory do other sorting algorithms require?
- \( N + O(1) \) for insertion sort and selection sort.
- In-place = \( N + O(\log N) \).

**Challenge for the bored.** In-place merge. [Kronrud, 1969]
Mergesort analysis: Comparison count

**Def.** \( T(N) \) = number of comparisons to mergesort an input of size \( N \)

\[
T(N) = T(N/2) + T(N/2) + N
\]

left half

right half

merge

Mergesort recurrence \( T(N) = 2 \cdot T(N/2) + N \) for \( N > 1 \), with \( T(1) = 0 \)

- not quite right for odd \( N \)
- same recurrence holds for many algorithms
- same number of comparisons for *any* input of size \( N \).

Solution of Mergesort recurrence \( T(N) \approx N \log_2 N \)

- true for all \( N \), as long as integer approx of \( N/2 \) is within a constant
- easy to prove when \( N \) is a power of 2.

\( \log N = \log_2 N \)

Mergesort recurrence: Proof 1 (by recursion tree)

\[
T(N) = 2 \cdot T(N/2) + N
\]

(assume that \( N \) is a power of 2)

\[
T(N) = N \log N
\]

Mergesort recurrence: Proof 2 (by telescoping)

\[
T(N) = 2 \cdot T(N/2) + N
\]

(assume that \( N \) is a power of 2)

Pf.

\[
T(N) = 2 \cdot T(N/2) + N
\]

(assume that \( N \) is a power of 2)

\[
T(N) = N \log N
\]

Mergesort recurrence: Proof 3 (by induction)

**Claim.** If \( T(N) \) satisfies this recurrence, then \( T(N) = N \log N \).

Pf. [by induction on \( N \)]

\[
T(N) = 2 \cdot T(N/2) + N
\]

(assume that \( N \) is a power of 2)

**Base case:** \( N = 1 \).

**Inductive hypothesis:** \( T(N) = N \log N \)

**Goal:** show that \( T(2^k) = 2N \log (2^k) \).

\[
T(2^k) = 2T(2^{k-1}) + 2N
\]

given

\[
= 2N \log N + 2N
\]

inductive hypothesis

\[
= 2N \log (2^k) - 2N + 2N
\]

algebra

\[
= 2N \log (2^k)
\]

QED

Ex. (for COS 341). Extend to show that \( T(N) \approx N \log N \) for general \( N \)
Bottom-up mergesort

Basic plan:
- Pass through file, merging to double size of sorted subarrays.
- Do so for subarray sizes $1, 2, 4, 8, \ldots, N/2, N$.

proof 4 that Mergesort uses $N \log N$ compares

```
public class Merge
{
   private static void merge(Comparable[] a, Comparable[] aux, int l, int m, int r)
   {
      for (int i = l; i < m; i++) aux[i] = a[i];
      for (int j = m; j < r; j++) aux[j] = a[m + r - j - 1];
      int i = l, j = r - 1;
      for (int k = l; k < r; k++)
         if (less(aux[j], aux[i])) a[k] = aux[j--];
         else a[k] = aux[i++];
   }

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

Concise industrial-strength code if you have the space

Mergesort: Practical Improvements

Use sentinel.
- Two statements in inner loop are array-bounds checking.
- Reverse one subarray so that largest element is sentinel (Program 8.2)

Use insertion sort on small subarrays.
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $N \approx 7$ elements.

Stop if already sorted.
- Is biggest element in first half $\leq$ smallest element in second half?
- Helps for nearly ordered lists.

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.

See Program 8.4 (or Java system sort)

Sorting Analysis Summary

Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

<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>

Lesson 1. Good algorithms are better than supercomputers.
**QuickSort**

**Basic plan.**
- **Shuffle** the array.
- **Partition** array so that:
  - element $a[i]$ is in its final place for some $i$
  - no larger element to the left of $i$
  - no smaller element to the right of $i$
- **Sort** each piece recursively.

**Quicksort Partitioning**

Q. How do we partition in-place efficiently?
A. Scan from right, scan from left, exchange

**Quicksort example**

Sir Charles Antony Richard Hoare
1980 Turing Award
public class Quick
{
    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int l, int r)
    {
        if (r <= l) return;
        int m = partition(a, l, r);
        sort(a, l, m - 1);
        sort(a, m + 1, r);
    }

    private static int partition(Comparable[] a, int l, int r)
    {
        int i = l - 1;
        int j = r;
        while (true)
        {
            while (less(a[++i], a[r]))
                if (i == r) break;
            while (less(a[r], a[--j]))
                if (j == l) break;
            if (i >= j) break;
            exch(a, i, j);
        }
        exch(a, i, r);
        return i;
    }

    public static void main(String[] args)
    {
        String[] a = {"apple", "banana", "cherry", "date", "elderberry"};
        sort(a);
        for (String s : a) System.out.println(s);
    }
}

Quicksort Implementation details

Partitioning in-place. Using a spare array makes partitioning easier, but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The (i == r) test is redundant, but the (j == l) test is not.

Preserving randomness. Shuffling is key for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to partitioning element.
Theorem. The average number of comparisons $C_N$ to quicksort a random file of $N$ elements is about $2N \ln N$.

- The precise recurrence satisfies $C_0 = C_1 = 0$ and for $N \geq 2$:
  
  $C_N = N + 1 + \left( (C_0 + C_{N-1}) + \ldots + (C_{k-1} + C_{N-k}) \right) / N$
  
  $= N + 1 + 2 \left( C_0 + \ldots + C_{k-1} + \ldots + C_{N-1} \right) / N$

- Multiply both sides by $N$:
  $NC_N = N(N + 1) + 2 (C_0 + \ldots + C_{k-1} + \ldots + C_{N-1})$

- Subtract the same formula for $N-1$:
  $NC_N - (N - 1)C_{N-1} = N(N + 1) - (N - 1)N + 2 C_{N-1}$

- Simplify:
  
  $NC_N = (N + 1)C_{N-1} + 2N$

Divide both sides by $N(N+1)$ to get a telescoping sum:

$$C_N / (N + 1) = C_{N-1} / N + 2 / (N + 1)$$

$$= C_{N-2} / (N - 1) + 2/N + 2/(N + 1)$$

$$= C_{N-3} / (N - 2) + 2/(N - 1) + 2/N + 2/(N + 1)$$

$$= 2 \left( 1 + 1/2 + 1/3 + \ldots + 1/N + 1/(N + 1) \right)$$

Approximate the exact answer by an integral:

$$C_N \approx 2(N + 1) \left( 1 + 1/2 + 1/3 + \ldots + 1/N \right)$$

$$= 2(N + 1) \ln N \approx 1.39 N \ln N$$

Quicksort: Summary of performance characteristics

Worst case. Number of comparisons is quadratic.
- $N + (N-1) + (N-2) + \ldots + 1 = N^2 / 2$.
- More likely than your computer is struck by lightning.

Average case. Number of comparisons is $\sim 1.39 N \ln N$.
- 39% more comparisons than mergesort.
- But faster than mergesort in practice because of lower cost of other high-frequency operations.

Random shuffle
- Probabilistic guarantee against worst case
- Basis for math model that can be validated with experiments

Caveat emptor. Many textbook implementations go quadratic if input:
- Is sorted.
- Reverse sorted.
- Has many duplicates (even if randomized) [stay tuned]

Sorting analysis summary

Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

<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>

<table>
<thead>
<tr>
<th>Quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>thousand</td>
</tr>
<tr>
<td>instant</td>
</tr>
<tr>
<td>instant</td>
</tr>
</tbody>
</table>

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.
Quicksort: Practical improvements

Median of sample.
- Best choice of pivot element = median.
- But how to compute the median?
- Estimate true median by taking median of sample.

Insertion sort small files.
- Even quicksort has too much overhead for tiny files.
- Can delay insertion sort until end.

Optimize parameters.
- Median-of-3 random elements.
- Cutoff to insertion sort for 10 elements.

Non-recursive version.
- Use explicit stack.
- Always sort smaller half first.

All validated with refined math models and experiments

Insertion sort animation

Mergesort animation

mergesort
mergesort analysis
quicksort
quicksort analysis
animations