Sorting and Sorting Algorithms

- Common computational task, embedded in many systems and requiring various types of performance.
- Rich variety of algorithms illustrating:
  - general schemes for constructing/inventing algorithms
  - a range of complexities, performances, and behaviours
  - simple correctness arguments

See: Chapter 8 of course textbook.

What is sorting?

Task: To rearrange a list of items which support a total order operation into ascending (i.e. non-descending) order.

More precisely:

Definition. An *order relation* ('comparison') is a binary relation $\leq$ on a set A satisfying:

- Reflexive: $x \leq x$ for all $x \in A$
- Antisymmetric: $x \leq y$ and $y \leq x$ implies $x = y$ for all $x, y \in A$
- Transitive: $x \leq y$ and $y \leq z$ implies $x \leq z$ for all $x, y, z \in A$.

If in addition the following property holds, the relation is called a *total order*:

- Dichotomy: $x \leq y$ or $y \leq x$ for all $x, y \in A$.

Specification

Sorting is a function: sort : list$[A] \rightarrow$ list$[A]$ where list$[A]$ is the type of finite linear lists of elements of type A, where A supports a total order $\leq$.

To be a sorting function it must satisfy:

perm(s,sort(s)) and ord(sort(s)) for all s : list$[A]$

the first condition says that the output is a *permutation* (ie rearrangement) of the input, the second condition says that the output is *ordered into ascending order*.

Definitions

$\mathrm{ord}(s) = \forall i, j \in \mathit{indices}(s). i \leq j \Rightarrow s[i] \leq s[j]$

$\mathrm{perm(s,t)} = \forall x \in \mathit{items}(s) \cup \mathit{items}(t).\ \mathit{count}(x, s) = \mathit{count}(x, t)$

where $\mathit{count}(x, s)$ is the number of occurrences of an item x in a list s.

Sorting Algorithms

Where do algorithms come from? How do we devise algorithms for tasks?

One source arises through *general schemes* for constructing algorithms. There are a range of such schemes, which if applicable, provide not just algorithms but an analysis of correctness and performance.

An example:

Divide and Conquer

The scheme is: *Divide* the input into parts, *solve the parts* (often recursively), then *combine* the solutions to give the final result.

Picture:

There are two extreme forms of this scheme:

- All the 'work' of the algorithm is in the **combine** stage, with a simple splitting of the input.
- All the 'work' of the algorithm is in the **division** of the input, it being so divided that the combine stage is then simple.

Mergesort

This is an example of a divide-and-conquer sorting algorithm with all the comparison operations in the combine stage:

1. split the input list arbitrarily into two lists, whose elements are exactly those of the original list,
2. *recursively* sort the two parts,
3. *merge* the resulting ordered lists into the final ordered list.

Of course, for all such recursive algorithms there are some base cases to consider.

Program for Mergesort

```
void mergesort(int a[], int low, int high)  {
  if(low == high)
       return;
  int length = high-low+1;
  int middle = (low+high) / 2;
  mergesort(a, low, middle);
  mergesort(a, middle+1, high);
  merge(a, low, middle, high);
}
```

We could have written this in functional style, returning a result list. However, here we've returned the result in the original list.

Notice the **divide function** is embedded in the program as the simple division of the input at a 'middle' point of an array, taking left and right sides of the middle as the two lists. Other ways of dividing the list (eg as even positions, and odd positions) give other versions of this algorithm and can be advantageous (eg for linked lists, where access to middle items is expensive).

The **merge method** for merging two lists in ascending order to produce a result in ascending order is, in principle, straightforward:

The base cases are simple. Otherwise, compare the head of both lists, the smaller is the head of the result, remove this element, and recursively merge the remaining lists,

[Exercise: if you haven't seen it before, encode this merge in the recursive style suggested, and in an iterative style, in both cases you will need to keep track of elements as they move around the array.]

Correctness of Mergesort

If we use the notation `a[x..y]` for the part of the array `a` from positions `x` to `y` inclusive.

The proof proceeds by induction. The base cases of 0 and 1 elements are straightforward. Now assume that the algorithm is correct on all lists of shorter `length`.

In particular, both `mergesort(a, low, middle)` and `mergesort(a, middle+1, high)` return permutations of that part of the list and in ascending order.

Then we need the crucial correctness condition for the `merge` operation: If `ord(a[low..x])` and `ord(a[x+1..high])`, then after `merge(a, low, x, high)` is called, `ord(a[low..high])` holds, and the result is a permutation of the input list.

We prove this again by induction, and then the correctness of mergesort follows.

Complexity of Mergesort

Unlike the correctness, the complexity depends crucially on the evenness of the splitting of the input list.

What operations do we count? The usual operation which indicates running time is the order operation comparing two elements.

For an approximately even split, the worst-case time complexity on list of length N is given by the recurrence relation:

$$C_N = 2 \times C_{N/2} + N$$

because the two recursive calls are on lists of length approximately $N/2$ and the merge function in the worst case requires approximately N comparisons - it is a linear merge (why?).

The *solution* to this (given previously) is a lin-log behaviour:

$$C_N = O(N \times \log(N))$$

Optimal Algorithm

In terms of worst-case time complexity, **Mergesort is optimal**. That is, it is the best possible worst-case performance for general sorting algorithms. The proof of this may be found in textbooks.

The average-case performance is the same (as is the best case!).

————————-

However, Mergesort *cannot* be implemented as an IN-PLACE algorithm, using only the input array to perform the sorting. It requires copying of the array which can be expensive for large tasks.

The Insertsort Algorithm

What happens to *Mergesort* if the split is very uneven?

Suppose for instance that we simply divide the head of the list from the tail to form two lists. Then we get a new algorithm, which is called *Insertsort*:

- If the list is empty, return empty list.
- Otherwise, remove head from list, *recursively* sort the tail using *Insertsort*, and then *Insert* the head item back into the list in the correct position.

To *Insert* an item into a list already in ascending order to return a list in ascending order:

- If the list is empty, return the singleton consisting of the inserted item.
- Otherwise, *compare* head with item to be inserted, if item is

less than head, then insert it at the front of the list, otherwise *recursively* insert item into the tail of the list.

Correctness

Can construct correctness argument again by induction, but it is a special case of the argument for Mergesort.

Performance

Both the worst-case and the average-case time complexity are given by:

$$C_N = C_{N-1} + N.$$

The solution is $C_N = O(N^2)$. This is considerably worse (much, much slower) than Mergesort, and illustrates how an uneven split can change the performance considerably.

However, it can be implemented as an IN-PLACE algorithm, so is space efficient.

Program for Insertsort

Insertsort may be implemented recursively as described above, or iteratively (using loops) by assembling the list item by item, inserting each time into the items already considered and arranged in ascending order:

```
void insertsort(int a[]) {
   for (int i = 1; i < a.length; i++) {
     int j = i;
     int B = a[i];
     while ((j > 0) && (a[j-1] > B))
       { a[j] = a[j-1]; j--; }
     a[j] = B;
   }
 }
```


Another range of recursive sorting algorithms arises from divide-and-conquer by making the combine stage easy, and putting all the work into the divide stage. An example is...

Quicksort

The easiest method of combining two lists is by *concatenation* (sticking them together end-to-end).

Question: How can we ensure that for two lists in ascending order, their concatenation is in ascending order?

Answer: A necessary and sufficient condition is that all items in the first list are less-than-or-equal-to all items in the second list.

So... this condition is what we need to ensure the split/division satisfies for the correctness of an algorithm. How do we do this?

Pivot elements

Idea: Choose an element (not necessarily in the list to be sorted), call it the pivot element. Then separate the list into those items less than (or equal to) the pivot element, and those items greater than the pivot element.

We can then sort the list, by *recursively* sorting these two sublists and then concatenating them for the result in ascending order.

Algorithms in this form are called *Quicksort* (Tony Hoare 1965).

```
void quicksort(Comparable a[],int left,int right)
  { if (left<right)
    { int x = split(a, left, right); // divides list
      quicksort(a,left,x-1);         // at x
      quicksort(a,x+1,right);
     }
   }
```


The natural way to implement the split function is with two additional arrays, but in the program below we improve this with an in-place implementation.

```
int split(Comparable a[], int left, int right)
   { Comparable pivot = a[left]; int x = left;
     for (int r = left+1; r <= right; r++)
      { if (a[r].compareTo(pivot) < 0)
        {a[x] = a[r];a[r] = a[x+1];a[x+1] = pivot; x++;}
      }
    return x;
   }
```


Correctness of Quicksort

By induction: When `left` is greater than or equal to `right` (ie array is empty or has one element) nothing is done.

Now suppose that `quicksort(a, left,x-1)` and `quicksort(a, x+1,right)` are sorted versions of those parts of the array. The split function ensures that all elements to the left of the pivot are less that or equal to the pivot element, and those to the right are greater than the pivot. Hence the resulting array is a permutation in ascending order.

Performance of Quicksort

The performance **depends on the eveness of the splitting** of the list. Unlike Mergesort, Quicksort doesn't determine the eveness in advance, but it depends upon the value of the pivot.

The **worst-case** is when all elements are either greater than, or less than, the pivot elements. The list is then split $1 : (N-1)$.

So $C_N = C_{N-1} + N$ because the splitting takes approx N comparisons.

Hence $C_N = O(N^2)$.

This is a very poor performance in the worst-case. Note: This is the case if the input is already in ascending or descending order!

If *the split is fairly even*, then

$$C_N = 2 \times C_{N/2} + N.$$

So $C_N = O(N \times \log(N))$ and in this case it performs well. This is possibly an **average-case** depending upon what kind of lists are likely to arise as input.

Note: The choice of pivot element affects time complexity but not correctness.

Better choices of pivot may ensure behaviour closer to $O(N \times \log(N))$ in more cases. For example: choose the mean (average) of first and last elements, or an average of three... choice depends on application.

Another possibility: Choose the elements at random, ie randomize the list before sorting it. Such algorithms which attempt to reduce bias by making random choices are called **Sherwood algorithms** (after Robin Hood!).

The Selectsort Algorithm

A special case of Quicksort is when the division of the N elements is $1 : (N-1)$. This is **Selectsort** and operates by repeatedly selecting and extracting the minimum (or maximum) element of the list.

Its worst-case and average-case time complexity is clearly $O(N^2)$.

Operation Decomposition

Another general scheme for constructing algorithms:

Idea: Consider the major operation(s) involved in the task. Decompose them into a combination of simpler operations.

For sorting, this leads to a family of algorithms called...

Interchange Sorting Algorithms

Idea: Decompose the rearrangement (permutation) operation on items in the list into a sequence of interchanges of *pairs* of items (these interchanges are sometimes called *transpositions*). Pairs may be adjacent or not.

Proposition: Any permutation can be constructed as a sequence of transpositions of adjacent elements.

Thus all we need to do is choose a sequence of transpositions so that the result is a list in ascending order. One example is **Bubblesort**:

```
void bubblesort(int A[])
 { for (int i = A.length; --i>=0;)
     {   boolean swapped = false;
         for (int j = 0; j<i; j++)
             { if (A[j] > A[j+1])
               { int T = A[j];
                 A[j] = A[j+1];
                 A[j+1] = T;
                 swapped = true; } }
             if (!swapped) { return; } } }
```

[Give correctness argument.]

The time complexity of Bubblesort is clearly $O(N^2)$, in worst-case and in average-case too.

But it is an IN-PLACE algorithm so is space efficient.

Proposition: Any sorting algorithm that operates by interchanging *adjacent* elements has an average-case time complexity of $O(N^2)$ or worse.

So-called **Shell sorts** which interchange non-adjacent pairs of elements can be designed to have complexities of order $O(N^{3/2})$!

Specialist Sorting Algorithms

So far, all sorting algorithms work on any items that support a total order operation.

However, if we consider *particular types of elements*, then we may devise specialist sorting algorithms which are more efficient than the general algorithms because they operate using more than just comparisons. As an example:

Bucketsort

This sorts lists of integers in the range $0, \ldots, M-1$ by an obvious method:

Algorithm: Create a sequence B of M sequences of integers, initially set to empty sequences.

To sort sequence A:

1. Scan A from left to right putting $A[i]$ into the sequence $B[A[i]]$,
2. Concatenate the resulting sequences of B in order.

Performance: There are no comparisons, but N items each handled once so $O(N)$, but space requirement can be large.

This is a special case of **Radix sort** algorithms, which rely on a *multi-way split* rather than comparison operations. They can be used, for example, for sorting strings.

Sorting using Trees

We return to general sorting algorithms.

Idea: Store the result of each comparison, not in a sequence, but in a tree, and then form a sorted sequence from this tree.

Various forms of tree are available. We give an algorithm based on *ordered binary trees*.

Example of binary trees in Java:

```
class BinaryNode
{   Comparable element;   // Label at each node
    BinaryNode left;      // Left subtree
    BinaryNode right;     // Right subtree
        // Constructors:
    BinaryNode( Comparable theElement )
      { this( theElement, null, null ); }
    BinaryNode( Comparable theElement,
                BinaryNode lt, BinaryNode rt )
      { element = theElement;
        left    = lt;
        right   = rt;
      }
   }
```


Type Invariant: A binary tree is an *ordered tree* if it satisfies the following type invariant (a predicate on the data type which is preserved under the operations):

1. The empty tree is ordered.
2. A tree consisting of a left subtree l a node labelled n and a right subtree r is ordered, just when
   - all nodes in the left subtree have labels less than n, and
   - all nodes in the right subtree have labels greater than or equal to n, and
   - the left and right subtrees l and r are both ordered.

Treesort

To use trees to perform a treesort, we begin by constructing ordered binary trees (these are standard operations on ordered binary trees).

Inserting an element in an ordered binary tree (element is inserted at a leaf of the tree):

```
BinaryNode insert(Comparable x, BinaryNode t)
  { if (t == null)
        t = new BinaryNode( x, null, null );
      else if (x.compareTo( t.element) < 0 )
        t.left = insert( x, t.left );
      else if (x.compareTo( t.element) > 0 )
        t.right = insert( x, t.right );
      return t; }
```


An *in-order* printing of a tree:

```
void printTree(BinaryNode t)
 {  if (t != null)
      { printTree( t.left );
        System.out.println( t.element );
        printTree( t.right );
      } }
```

Now, to sort a list into ascending order:

1. Insert each item of the list into a tree, starting with the empty tree, to form an ordered binary tree,
2. Output items from tree using the in-order traversal.

Correctness of Treesort: The correctness is direct from the type invariant for ordered trees using list and tree induction.

Performance of Treesort: The comparison operations take place in the insert function only. Each insert starts at the root and traverses to a tip of the tree - so the number of comparisons is related to the height of the tree.

Worst-case time complexity: If the tree is entirely unbalanced, ie consists of one path only: In this case the number of comparisons is $1 + 2 + 3 \cdots + N = O(N^2)$. Poor performance in worst-case. This is the case if the input list is already ordered!

If the tree is approximately balanced then the height of the tree is approximately $\log_2(N)$ for N nodes.

Thus to insert N nodes in a balanced tree is approximately $O(N \times \log_2(N)$. (Actually this is quite accurate: we need to calculate the sum $\sum_{i=0}^{N}$ ceiling$(log_2(i+1))$.)

The tree traversal to output the items in ascending order is $O(N)$, so overall this is an $O(N \log(N))$ sorting algorithm if the tree remains fairly balanced. This represents an average-case if input lists are arranged randomly.

This is still *space inefficient*: we have to create the tree as an intermediate structure.

Tree balancing techniques

To improve performance, we can try to *maintain a balanced tree*. There are a variety of techniques for moving nodes around trees to balance them: See second part of the module.

Finally, a question:

Question: Is there a general sorting algorithm that is $O(N \log(N))$ in the worst-case and is an in-place algorithm? We have not met one yet!

Answer: Yes, **Heapsort** is such an algorithm, and uses a datatype called a *Heap* (or *Priority Queue*). See textbooks for details.