Algorithms and Data Structures
Week 4: Sorting

Abstract

Summary: This week we'll be looking at sorting algorithms on list-like structures.

- **Book reference**: Drozdek chapters 9.3.4–9.3.5;
- **Resources**: Week 4 archived eclipse program bundle.
- **Group exercises to hand in:** In your groups, complete exercises 1, 2, 6 and 12.

  There is a submission box open in iLearn for these solutions. **Submission is entirely optional, but even if you do not submit it is strongly recommended that you attempt the questions.** Although there are no marks for submission, recall that the Week 6 in class test will be formed from a selection of the exercises you have been asked to study in these workshops.
- **Extension question:** Exercises 8 and 13. (These are for extension only: they will not be part of the test.)

Sorting

We'll look at a number of sorting algorithms implemented on arrays. We'll concentrate particularly on using invariants to make sure we get the implementations and designs right and on using complexity analysis to reflect on the resource usage of the various designs.

Mergesort on arrays

The basic idea of mergesort is to notice that if we have two sorted arrays then it's pretty easy to put them together to get one big sorted array containing all the elements of the two arrays "merged" into one. This leads to the following recursive strategy:

```
void mergeSort(char A[], int first, int last){
 // POST: sorts the array portion A[first..last]
       if (first < last) { // just swap the nodes;
         mergeSort (A, first, (first+last)/2);
         mergeSort (A, (first+last)/2 + 1, last);
       // PRE for MERGE: the array portions A[first..(first+last)/2]
       //                and A[(first+last)/2+1..last] are sorted
       merge(A, first, (first+last)/2+1, last);
      }
  }
```

The specification of merge

```
void merge(char A[], int a, int mid, int b);
   // PRE: the array portions A[a..mid-1]
    //     and A[mid..b] are sorted
    // POST: The array portion A[a..b] is sorted
```

The implementation for `merge` uses an auxiliary array to copy items from `A` to merge the two portions together. The final step then copies all the items back to `A`. This results in an additional space overhead, which could be considerable for large arrays.

Design issues for arrays

Simplicity of the implementation: relatively easy on arrays $T(n) = 2T(n/2) + O(n)$, giving a time complexity of $O(nlog(n))$.

However merge uses auxiliary arrays and so needs a lot of space: $O(n)$ of temporary storage.

Quicksort on arrays

Quicksort is a well-known sorting algorithm which behaves very efficiently on most input arrays. Here's the idea. An item is in correct sorted order if it would not change position were the array to be fully sorted. This means an array is sorted if *all* items are in correct sorted order. Quicksort gradually puts items in correct sorted order as follows.

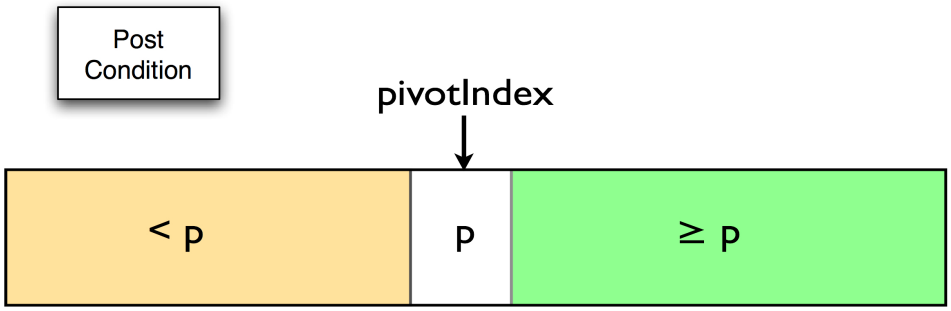
- Gradually place items in their correct sorted order until they are all in their correct sorted order.
- Any item is in its correct sorted order then its position will not change were the array to be sorted. This means that if all items are in their correct sorted order then the array must be sorted!
- If an item is such that all items to the left are smaller, and all items to the right are greater, then it must be in correct sorted order! This property gives us an easy way to check for correct sorted order without actually sorting the whole array.

Here's the pseudo code which relies on the correct implementation of `partition`.

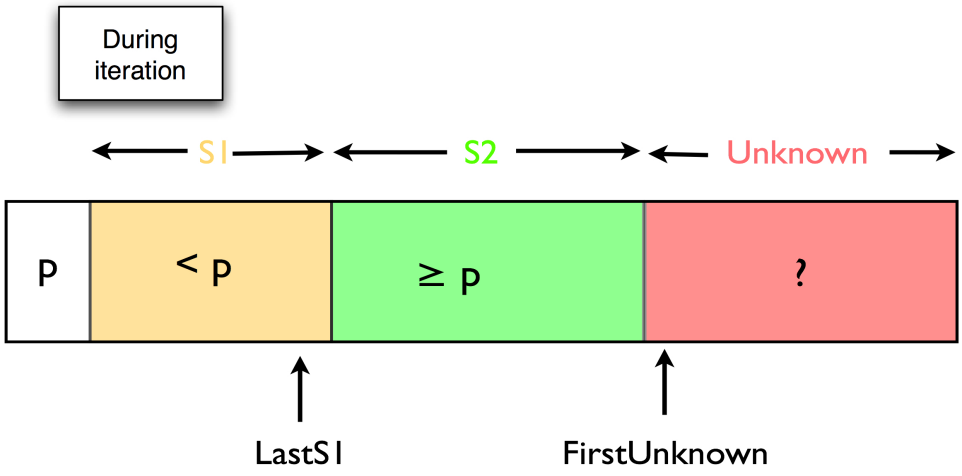
```
void quicksort (int A[], int first, int last) {
   if (first < last) {
     1. select an array item p at random in A[first, last];
     2. swap p into the index first;
     3. call partition(A,  first, last); pivotIndex is set to the result.
     4. quicksort(A, first, pivotIndex-1);
     5. quicksort(A, pivotIndex +1, last);
```

```
   }
}
int partition(int A[], int first, int last);
// POST: takes the item p=A[first] and places it in its correct
// sorted position, rearranging the array so that all items to the
// left of p are no more than p, and all items to the right of it are
// no less than p.  The return value (pivotIndex in quicksort) is the index of p in its
// correct sorted order.
```

The idea for `partition` is to group the array into three sections, one of which is an element containing the distinguished value `p`; the other two portions are to the left of `p` which should all be less then `p` and the portion to the right of `p` where all items should have value at least `p`. The post condition gives a pictorial representation of this goal.



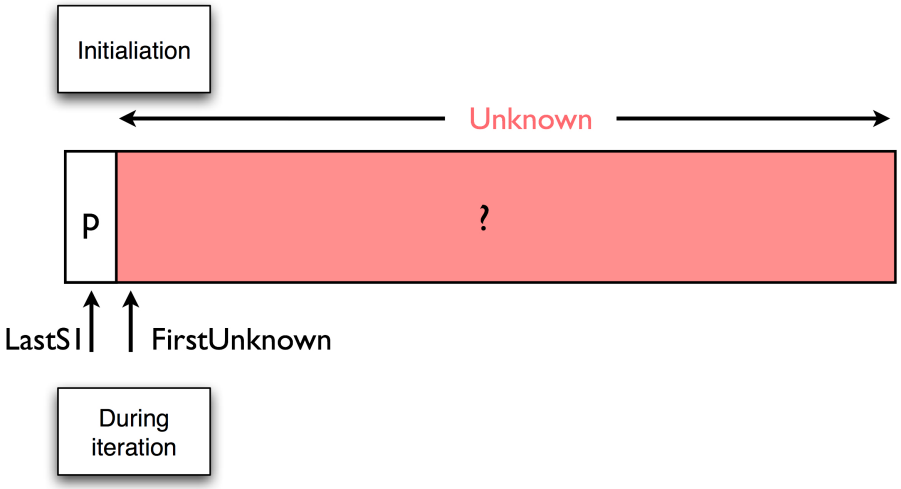
The strategy to achieve this is to create three regions of the array so that the first region contains items less than `p`, the second region contains items at least `p`, and the third region is unexplored or Unknown.



As we investigate the Unknown region, we are able to iteratively add items to `S1` (less than `p`) or `pS2` (greater than `p`), until the Unknown region is emptied. The second representations of the array give pictorial representations of this idea.

Once the Unknown region is emptied so that all items have been swapped into either `S1` or `S2` then the item `p` can be swapped into its proper position.

To start the process off we need to ensure that the array is correctly initialised: i.e. before any of the array has been explored the entire array except for the first item is designated UnKnown.



In lectures we'll design the function `partition` so that it satisfies the invariant:

The items in region `S1` (delimited by `LastS1`) are all less than `p`, and the items in region `S2` (delimited by `FirstUnknown`) are at least `p`.

Or, for those who prefer a formula:

$(\texttt{first} < \texttt{i} <= \texttt{LastS1} \Rightarrow \texttt{A[i]} < \texttt{p}) \wedge (\texttt{LastS1} < \texttt{i} < \texttt{FirstUnknown} \Rightarrow \texttt{A[i]} \geq \texttt{p})$

Analysis of quicksort

Well count the number of comparisons.

For the partitioning, we need to do approximately (`last-first`) comparisons; and we do the partitioning for each recursive call.

In the case that we are unlucky with the choice of `p`, we may end up with one of regions `S1` or `S2` being empty. If that keeps happening then we will have about `N` recursive calls (where `N` is the size of the original array), with respectively `N, N-1, N-2 ....` etc. work for each partitioning, as above. An already sorted array is an example for this worst-case behaviour. (Why?) This makes the worst case behaviour $O(N^2)$.

In practice the worst-case is very unlikely to happen however, which is why it deserves its name!

Bucket sort

Bucket sort is appropriate when you know that there is a fixed number k of different values in e.g. an array.

Bucket sort maintains an array of counters, one for each of the k items, and increments the appropriate counter whenever it encounters an array item of the corresponding type of item.

Let `B[k]` be an array for the buckets, initialised everywhere to 0.

```
1. For array index i,
2.  if A[i] == m, then increment B[m];
3. After this scan, copy into A, the first B[0] indices as 0,
    the next B[1] indices as 1 etc.
```

Time: This requires one pass of the array of length n so it has complexity O(n).

Space: Uses scratch space the size of the number of fixed items.

Difficulty of design: Straightforward.

Limitations: not a general sort.

Radix sort

This kind of sort is appropriate when the items to be sorted may be characterised by a fixed number of independent attributes.

For example, a in pack of playing cards, each card may be identified by a suit (Diamonds, Hearts etc.) and a rank (Ace, Jack, King etc.), or any integer less than 1000 in decimal notation may be identified by three digits.

The idea of radix sort is to group items according to each of the (ordered) attributes in turn. Once combined after the grouping, the items are ordered as a whole.

Sorting a pack of cards

Cards in a regular set of playing cards have a suit and a rank. We order both separately:

$$\textbf{Suit} : \clubsuit < \diamondsuit < \heartsuit < \spadesuit$$

, and

$$\textbf{Rank} : 2 < 3 < \ldots < \mathit{Jack} < \mathit{Queen} < \mathit{King} < \mathit{Ace}$$

So now any card with a suit, rank pair (s, r) is regarded as being lesser than another (s', r') if either $s < s'$, or if $s = s'$ then $r < r'$. For example the card $(\diamondsuit, \mathit{King})$ is less than $(\spadesuit, 2)$.

We say that the rank is the least significant attribute (since it only plays a part in ordering within the suits, which is therefore the most significant attribute).

A radix sort of the cards first sorts the cards into piles according to the least significant rank, and then sorts the piles according to the more significant rank.

1. First sorts into 13 piles according to 2's, 3's etc.
2. Next for each pile sort first $\clubsuit$, then $\diamondsuit, \heartsuit, \spadesuit$.
3. Finally group the piles.

Radix sort of 3 digit integers

In decimal notation, each number is similarly lexicographically ordered according to the digits used in the notation:

$$0 < 1 < 2 < 3 < 4 < 6 < 7 < 8 < 9$$

But we can think of the order on three-digitted numbers as follows:

$$abc \leq abc$$
iff

$$a < a, \vee$$
$$a = a \wedge b < b, \vee$$
$$a = a \wedge b = b \wedge c \leq c$$

1. Group the numbers according to the least significant digit (ie the units), ordering the groups according to the standard ordering on digits;
2. Group the numbers according to the next significant digit (ie the 10s), ordering the groups according to the standard ordering on digits, making sure that the relative ordering according to step 1 is not disturbed.
3. Group the numbers according to the next significant digit (ie the 100s), ordering the groups according to the standard ordering on digits, making sure that the relative ordering according to steps 1 and 2 is not disturbed.

```
void radixSort (int A[], int n, int d)
// PRE: n is the length of A, d is the maximum number of digits
// of the integers in A
// POST: A is sorted
{ for (int j= 0; j < d 0; j++){  // sort each item according to digit
     set up 10 arrays, one for each group;
     set up ten counters, one for each group,
     each initialised to 0;
     for (int i= 0; i < n; i++) {// this is for each item
        place A[i] at the end of the appropriate
        group according to its jth digit;
        increment that groups counter;
      }
     copy all the items in the groups into A, starting
     with 0'th group, then 10'th etc.
   }
```

0.1 Analysis

For each digit j, there are 2n passes through items of A;

There are d digits, so we do that d times.

Thus it takes about 2*d*n.

This is O(n), where n is the length of the array (since d is a fixed constant, and assumed to be much less than n).

Issues:

Time: This requires one pass of the array of length n so it has complexity O(n).

Space: Uses scratch space the size of the number of fixed items.

Difficulty of design: Complicated to keep track of all the counters.

Limitations: not a general sort, not in-place, but can transfer well to linked lists.

1 Exercises

1. A company database consists of 10,000 sorted alphabetically names, 40% of whom are known as good customers and who together account for 60% of the accesses to the database. There are two data structure options to consider for representing the database:

   · Put all the names in a single array and use binary search.

   · Put the good customers in one array and the rest of them in a second array. Only if we do not find the query name on a binary search of the first array do we do a binary search of the second array.

   Give reasons to determine which option gives better expected performance. Does this change if linear search on an unsorted array is used instead of binary search for both options?

2. Suppose you are given an array A of n sorted numbers that has been circularly shifted k positions to the right. For example,

   $$\{35, 42, 5, 15, 27, 29\}$$

   is a sorted array that has been circularly shifted $k = 2$ positions, while

   $$\{27, 29, 35, 42, 5, 15\}$$

   has been shifted $k = 4$ positions.

   (a) Suppose you know what k is. Give an $O(1)$ algorithm to find the largest number in A.

   (b) Suppose you do not know what k is. Give an O(n) algorithm to find the largest number in A..

   (c) (Harder!) Suppose you do not know what k is. Give an $O(\log n)$ algorithm to find the largest number in A. [Hint: think "Divide and Conquer".]

3. For each problem below give a Java implementation, first for an unsorted array of integers, and second for a sorted array of integers. You must give some clear comments in your code to describe your design, and state the pre-condition and post-condition for any methods or procedures you would use.

   Give loop invariants for any loops you use, or clear reasoning why a loop might not be necessary. Estimate the worst case complexity for each with reasons. Provide some JUnit tests to check your work.

- Given an array A, give an algorithm that finds the pair i, j such that $|A[i] - A[j]|$ is maximum amongst all such pairs, where $|x - y|$ is the absolute value of $x - y$ for integers x, y.
- Given an array A, give an algorithm that finds the pair i, j with $i \neq j$ such that $|A[i] - A[j]|$ is minimum amongst all such pairs.

4. Trace the `merge(A, 0, 2, 4)` the array function for the array `A = {'b', 'd', 'a', 'c', 'e'}`. For the first while loop (ONLY), check that its invariant

```
// INVARIANT for all the loops:
// tA[a.. a + (i-a) + (j-mid) -1] is the sorted merge
//            of A[a..i-1] and A[mid..j-1]
```

   holds.

5. Now rewrite the `mergeSort` and `merge` functions so that they take an additional boolean parameter called `upDown`. If `upDown` is set it `true` then it performs an ascending sort, otherwise it performs a descending sort. Write a set of appropriate JUnit tests to check your programs.

6. Trace this partition function discussed in lectures for the array

$$\texttt{A} = \{27, 38, 12, 39, 27, 16\}.$$

   After each iteration of the loop check that the invariant is true.

7. The partition function used in QuickSort given in lectures requires a final swap of the pivot item p. Now think of writing an alternative algorithm based on the following invariant.

> "All the items with index at least F but less than b are less than or equal to p; all the items with index greater than c, but no more than L have value at least p; A[b] == p"

   Here F, L are the first and last indices of the array A, c is a counter that starts at L and should be decremented, and b is an index that should always point to the position in the array whose item is p. Your algorithm should not require the final swap.

   [Hint: Draw a picture to help you envisage the shape of the invariant.]

8. Suppose the items in an array take one of three distinct values. For example, the array could be:

$$[0, 2, 1, 0, 0, 2, 1, 1, 2, 2, 2, 0, 1] \ ,$$

where the values are all $0, 1, 2$.

Use the ideas from the partition function for quicksort to implement an algorithm that sorts the items in O(N), where N is the length of the list. Write some JUnit tests to check your work.

[Hint: Think of swapping elements out of the unknown region of the array. Keep three sections of the list: an initial portion from index 0 to index $f - 1$ contains 0's; the portion from index f to index $m - 1$ contains 1's and the portion from index b to $N - 1$ contains 2's. Now think about swapping items in the unknown portion (from index m to index $b-1$) into one of the three portions.]

9. Trace the radix sort algorithm for the following list of letters.

```
BCA, ABC, CAB, ACB, BAC, CBA
```

10. Suppose that during one pass of the radix sort algorithm to sort the elements for attribute `k`, it is found that the items were already sorted with respect to that attribute. Explain how that situation can be identified by inspecting the counters. Now adjust the implementation given in lectures for radix sort to take advantage of this observation to improve efficiency.

11. Let `A` be an unsorted array of integers. Using the `partition` function from `quicksort` design an efficient algorithm which finds the `k`'th smallest item in `A`. The k'th smallest item is the item which appears at index `k` were the array to be completely sorted. Your solution should not sort the array entirely — you can do it more efficiently than that.

    [Hint: Think about what would happen if your partition function returned `k`. Would you know the value of the `k`'th smallest item? if partition returned a value less than `k` which part of the array would contain the `k`'th smallest item?]

12. A sort is said to be *stable* if whenever two elements have the same value, then they must appear in the same order in the output as in the input. State which of `mergeSort`, `quickSort`, `bucketSort` and `radixSort` are stable. In the case of the unstable ones, give an example.

13. Design a stack that supports push, pop, and retrieving the minimum element in constant time. Can you do this? Give reasons.

    In the case that you can, describe the data structure and any invariants you need to be satisfied by the stack methods to ensure that your stack works correctly.