51 Sorting algorithms *(2 activities)*

51.1 Explore several sorting algorithms.(23 steps)

51.1.1 (Brainstorm) Sorting by hand (1)

Explain how you would sort a hand of 13 playing cards as you are dealt the cards one by one. Your hand should end up sorted by suit, and then by rank within a suit.

51.1.2 (Brainstorm) Sorting by hand (2)

Explain how you would sort a pile of 300 CS 61A exams by login name. If your algorithm differs from your card-sorting algorithm of the previous step, explain why.

51.1.3 (Display page) Overview of sorting algorithms

Overview of sorting algorithms

Today we'll explore several algorithms for *sorting* a collection of data. The methods we'll be working with will sort an array or linked list of integers, but can easily be adapted to work with a collection of `Comparable` objects. There are several categories of sorting algorithms:

- selection sorts, which repeatedly remove first the largest element, then the second largest, and so on from a collection;
- insertion sorts, which repeatedly insert elements into a sorted collection (your card sorting algorithm was probably one of these);
- exchange sorts, which repeatedly exchange elements that are out of order until the collection is sorted;
- merge sorting, in which sorted sequences of items are *merged* into larger sequences;
- distribution sorting, where elements are sorted into groups based on their "digits", and the groups are sorted and combined.

Activities for today will include coding and analysis. The simple algorithms generally perform *better* than their more complicated counterparts on small sets of data, and we'll be working with timing data to find out the crossover point at which the more complicated algorithms perform better.

51.1.4 (Display page) Insertion sort

Insertion sort

Insertion sort is basically an accumulation (recall accumulations from CS 3 and 61A?) that's coded in Scheme as follows:

```
(define (sorted L)
  (accumulate
    (lambda (x sortedSoFar) (insert x sortedSoFar))
    L) )
```

(You were briefly introduced to this algorithm back when you were learning about arrays and loop invariants.) Here is code that applies the insertion sort algorithm to an array named `values`.

```
for (int k=1; k<values.length; k++) {
    // Elements 0 ... k-1 are in order.
    int temp = values[k];
    int j;
    // Find the place to put the next element
    // while shifting elements down.
    for (j=k-1; j>=0; j--) {
        if (values[j]>temp) {  // comparison step
            break;
        } else {
            values[j+1] = values[j];
        }
    }
    // Put the new element in its proper place.
```

```
    // Elements 0 ... k are now in order.
    values[j+1] = temp;
}
```

51.1.5 (Brainstorm) Insertion sort analysis

Here's the insertion sort code.

```
for (int k=1; k<values.length; k++) {
    int temp = values[k];  int j;
    for (j=k-1; j>=0; j--) {
        if (values[j]>temp) {  // comparison step
            break;
        } else {
            values[j+1] = values[j];
        }
    }
    values[j+1] = temp;
}
```

Describe how to arrange the values in the array so that, when the above algorithm is applied, the number of comparisons done at the "`// comparison`" step is *minimized*. Then describe a worst-case arrangement of array values that *maximize* the number of executions of the "`// comparison`" step. Briefly explain your answers. Correct your answer if necessary after reviewing the responses of your labmates.

51.1.6 (Display page) Insertion sort applied to linked lists

Insertion sort applied to a linked list

The file `~cs61b/labcode/lab26/IntList.java` contains an implementation of a doubly-linked list, along with methods for sorting the list values. Supply the body of the `insert` method to complete the coding of the insertion sort algorithm.

51.1.7 (Display page) Selection sort

Selection sort

The selection sort algorithm, applied to a collection of `N` elements, involves the following loop:

```
for (int k=0; k<N; k++) {
    find and remove the largest element in the collection,
    and add it to the end of another sorted collection.
}
```

Given below are implementations of the selection sort algorithm for arrays and linked lists. Array sorting

```
for (int k=0; k<values.length; k++) {
    // Elements 0 ... k-1 are in their proper positions
    // in sorted order.
    int maxSoFar = values[k];
    int maxIndex = k;
    // Find the largest element among elements k ... N-1.
    for (int j=k+1; j<values.length; j++) {
        if (values[j] > maxSoFar) {
            maxSoFar = values[j];
            maxIndex = j;
        }
    }
    // Put the element in its proper place.
    // Elements 0 ... k are now in their proper positions.
    swap (values, maxIndex, k);
}
```

Linked list sorting

```
IntList sorted = new IntList ( );
```

```
while (myHead != null) {
    int maxSoFar = myHead.myItem;
    DListNode maxPtr = myHead;
    // Find the node in the list pointed to by myHead
    // whose value is largest.
    for (DListNode p=myHead; p!=null; p=p.myNext) {
        if (p.myItem > maxSoFar) {
            maxSoFar = p.myItem;
            maxPtr = p;
        }
    }
    sorted.addToEnd (maxSoFar);
    remove (maxPtr);
}
myHead = sorted.myHead;
```

One may observe that, in the first iteration of the loop, element 0 is compared with all the others. Then element 1 is compared with `N-2` other elements, element 2 is compared with `N-3` other elements, and so on. The total number of comparisons is `N-1 + N-2 + ... + 1 = N*(N-1)/2`, *no matter what the ordering of elements in the array or linked list prior to sorting*.

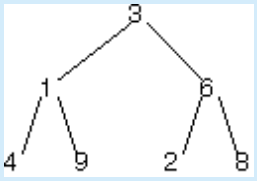
51.1.8 (Display page) Choosing a better sorting structure

Choosing a better sorting structure

In pseudocode, the selection sort algorithm is

```
for (int k=0; k<N; k++) {
    find and remove the largest element in the collection,
    and add it to the end of another sorted collection.
}
```

Adding something to the end of a sorted array or linked list can be done in constant time. Finding and removing the maximum element in a collection can also be done quickly if we organize the collection appropriately first. A max heap fills the bill; removal of the largest element from a heap of `N` elements can be done in time proportional to `log N`, so once the heap is created, sorting can be done in time proportional to `N log N`. (`N` removals from the heap are done, each taking time at worst proportional to `log N`.) An unexpected bonus is that we can organize `N` array elements into a heap in *linear* time. Here's an example of how to do this with an array containing 3, 1, 6, 4, 9, 2, 8. These values represent the tree



1. The leaves of the heap are 4, 9, 2, 8. Viewed in isolation, they are already heaps (each containing one element).
2. Now we process the internal heap nodes, starting at 6. Its children are 2 and 8, so it should trade places with the 8 to make the values 6, 2, 8 into a heap.
3. We move now to 1. Its children are 4 and 9; we exchange 1 with 9 to build a heap of those three elements.
4. Finally, we bubble 3 down.

Why is this faster? The number of levels each element moves down is at most its height. The sum of the heights of nodes in a binary tree of `N` nodes turns out to be proportional to `N`. Note: You may remember the heapify and heap sort algorithms being applied during lecture.

51.1.9 (Self Test) Building a heap in linear time

What array results from applying the linear-time heap creation algorithm to an array containing 5, 2, 3, 4, 6, 7?

51.1.10 (Display page) Choosing another good sorting structure

Choosing another good sorting structure

Insertion sort, in pseudocode, is

```
for each element in the collection to be sorted,
    insert it into its proper place in another collection.
```

The insertion sort algorithm we just considered did its insertion in an array, where elements had to be shifted over to make room for the new element. Choice of a structure that allows faster insertion—namely, a binary search tree—produces a faster algorithm. We build the tree through repeated insertions, then traverse it in linear time to produce the sorted sequence.

51.1.11 (Display page) "Divide and conquer"

"Divide and conquer"

Another way to speed up sorting is by using a "divide and conquer" technique:

1. Split the elements to be sorted into two collections.
2. Sort each collection recursively.
3. Combine the sorted collections.

Compared to selection sort, which involves comparing every element with every other, this dividing and conquering has the potential to reduce the number of comparisons significantly. Two algorithms that apply this approach are *merge sort* and *Quicksort*.

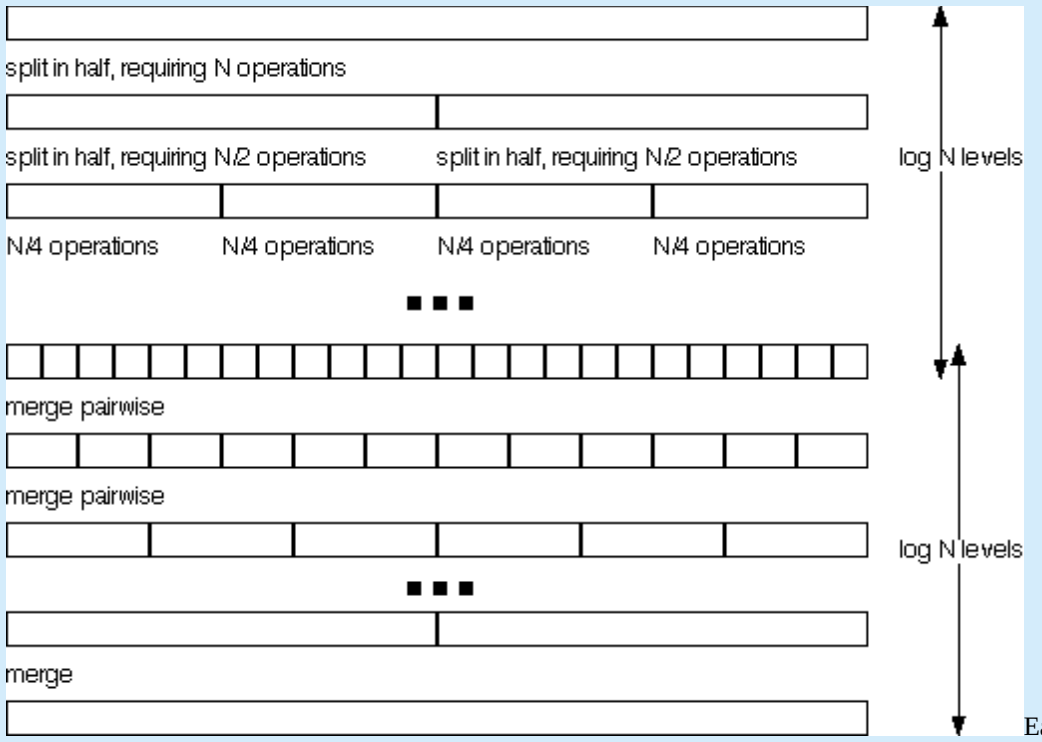
51.1.12 (Display page) Merge sort

Merge sort

Merge sort works as follows.

1. Split the collection to be sorted in half somehow.
2. Sort each half.
3. Merge the sorted half-lists.

Merging two sorted lists takes time proportional to the sum of the lengths of the two lists in the worst case. Splitting the collection in half requires a single pass through the elements. The processing pattern is depicted in the diagram below.



Each level in the diagram is a collection of processes that all together run in linear time. Since there are `2 log`$_2$ `N` levels, the total time is proportional to `N log N`.

51.1.13 (Display page) Quicksort

Quicksort

Another application of dividing and conquering appears in the Quicksort algorithm:

1. Split the collection to be sorted into two collections by *partitioning* around a "divider" element. One collection consists of elements greater than the divider, the other of elements less or equal to the divider.
2. Quicksort the two subcollections.
3. Concatenate the sorted large values with the divider, and then with the sorted small values.

Here's an example of how this might work, sorting an array containing 3, 1, 4, 5, 9, 2, 8, 6.

1. Choose 3 as the divider. (We'll explore how to choose the divider shortly.)
2. Put 4, 5, 9, 8, and 6 into the "large" collection and 1 and 2 into the "small" collection.
3. Sort the large collection into 9, 8, 6, 5, 4; sort the small collection into 2, 1; combine the two collections with the divider to get 9, 8, 6, 5, 4, 3, 2, 1.

Concatenation in an array or linked list can be done in constant time. If partitioning can be done in time proportional to the number of elements N, *and* it splits the collection more or less in half, this produces an algorithm that runs in time proportional to `N log N`. (Reasoning is similar to what we used for merge sort.)

51.1.14 (Brainstorm) Picking the divider

The median element would be the best divider. What would be the impact on Quicksort's running time if we partitioned the elements to be sorted by first finding the median, then using that value as the divider?

51.1.15 (Brainstorm) Worst case behavior?

Suppose that the first element is selected as the divider. Describe an arrangement of values in an array or

linked list that would produce the worst possible behavior of the Quicksort algorithm.

51.1.16 (Display page) Practical methods for selecting the divider

Practical methods for selecting the divider

Selection of the divider needs to be done quickly. Choosing the first element in the array or list is quick; a somewhat better choice when sorting an array is the median of the first, last, and middle elements. This avoids the worst case you came up with in the previous step.

51.1.17 (Display page) Quicksort a linked list

Quicksort a linked list

Some of the code is missing from the `quicksort` method in `~cs61b/labcode/lab26/IntList.java`. Supply it to complete the Quicksort implementation.

51.1.18 (Display page) Quicksort performance in practice

Quicksort performance in practice

Quicksort in practice turns out to be the best general-purpose sorting method. For example, it's the algorithm used in Java's `Arrays.sort` method. With some tuning, the most likely worst-case scenarios are avoided, and the average case performance is excellent. Here are some improvements to the basic Quicksort algorithm.

- When the number of items to sort gets small (the base case of the recursion), insertion sort is used instead.
- For large arrays, more effort is expended on finding a good divider element.
- Various machine-dependent methods are used to optimize the partitioning algorithm and the `swap` operation.

51.1.19 (Display page) Identifying mystery sorts

Identifying mystery sorts

This activity involves the use of a program called "Sort Detective" to identify "mystery" sort methods from their execution behavior. (Credit for Sort Detective goes to David Levine, Computer Science Department, St. Bonaventure University, New York.) We suggest that you do this with a partner. Start by copying the Sort Detective program to your own directory:

```
cp -r ~cs61b/labcode/lab26/sort.detective ~
```

Then `cd` to that directory and run Sort Detective:

```
cd ~/sort.detective
java SortDetective
```

The five buttons on the left correspond to the insertion sort, selection sort, merge sort, Quicksort, and heap sort algorithms. (The tested implementations are given in the file `sort.detective/allSorts.txt`) Your task is to find out which button goes with which sort. Create a list to be sorted by selecting relevant radio buttons and then clicking on "Create list". Then run one or more of the sorting methods on that list. Identify the sorting methods by observing their execution behavior on large and small lists, and already-ordered and randomly ordered lists.

51.1.20 (Brainstorm) Sorting out the sorts

Which button corresponds to which sort algorithm? Explain how you determined this.

51.1.21 (Display page) Timing the sort methods

Timing the sort methods

To finish off, we'll do some timing experiments. The program `~cs61b/labcode/lab26/sorting/SortPerf.class` applies one of a collection of sorting algorithms to randomly generated arrays whose size is provided as a command-line argument, and outputs

timing data to a file. An example run of `SortPerf` would be

```
java SortPerf select 5 175 200 select.dat
```

The first argument to `SortPerf` is the sorting method ("select", "merge", "heap", "insert" or "quick"), the second is the smallest array to sort and the increment between array sizes, the third is the maximum array size, and the last is the number of times to run each size. Accurate results require that each test be run numerous times, since elapsed time for individual runs will vary depending on what else is happening on the computer. The reported times are total milliseconds for all runs of a given size. The `java` command just given applies selection sort to arrays of size 5, 10, 15, ... , 175 and runs each one 200 times, placing the timing results in the file named `select.dat`. (The timings often look a little "bumpy"; that is, the running time does not appear to be strictly increasing with input size&mdash. Running `SortPerf` again sometimes helps.) Copy the directory `~cs61b/labcode/lab26/sorting` to your own directory, then `cd` to that directory. We have provided a shell script named `runrace` that you may use to run `SortPerf` for the same sizes on merge sort, insertion sort, Quicksort, heap sort, and selection sort; it creates data files named `insert.dat, select.dat, merge.dat, heap.dat,` and `quick.dat` Use a program called `gnuplot` to plot the timing results. At the prompt in a shell window (not `emacs` or Eclipse), type

```
gnuplot
```

When `gnuplot` has been loaded, you will see a ">" prompt. You can then plot the data produced by `runrace` in `gnuplot` using the command:

```
plot 'select.dat' with linesp 1, 'merge.dat' with linesp 2,
'insert.dat' with linesp 3, 'quick.dat' with linesp 4, 'heap.dat' with
linesp 5
```

Type the two lines above as one command, all on one line. You should observe that although the five algorithms asymptotically behave as expected, it is unclear which algorithm is faster for small array sizes. At what values of `N` do you observe crossovers in execution time between the five sorting algorithms? To help see these crossovers better, use the `runrace.2` shell script, then replot.

51.1.22 (Brainstorm) Timing results

Determine (roughly) where the crossover point is between the N^2 algorithms and the `N log N` algorithms.

51.1.23 (Display page) Animations

Animations

There are a lot of web sites that provide animations of sorting algorithms. Here are some good ones:

- http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html
  http://www.geocities.com/SiliconValley/Network/1854/Sort1.html
  These are patterned on the "race" animations in the "Sorting Out Sorting" video.
- http://www.cs.auckland.ac.nz/software/AlgAnim/qsort.html
  http://www.cs.auckland.ac.nz/software/AlgAnim/heapsort.html
  These animations of Quicksort and heap sort have lots of annotation.
- http://maven.smith.edu/~thiebaut/java/sort/demo.html
  This site has every algorithm we've considered today but merge sort.
- http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/mergeSort/mergeSort.html
  This animation of merge sort has lots of annotation.

51.2 Homework*(1 step)*

51.2.1 (Display page) Project 3

Project Work

For homework you should work on the project. In the last lab there was an additional checkoff assigned that is due before the next lab. (Below is a refresher of that checkoff. Please refer to yesterday's homework for the official list.) During the next lab, T.a.s will expect to see progress on your project. Evidence of progress should

include the following:

- a list of what you and your partner are doing;
- a description of what you have working so far;
- what, if any, bottlenecks you've encountered in your work so far.

This information would also be good to have in your README file. Among the things the t.a.s would like to see are several of the following:

- input of blocks from a file into a tray data structure;
- comparison of a tray and a goal configuration;
- generating moves from a given configuration;
- making a move;
- implementation of a good hashing scheme;
- a comprehensive `isOK` method for trays.

52 Quiz 25 *(1 activity)*

52.1 (Quiz) Analyze a graph algorithm.*(1 step)*

52.1.1 (Student Assessment) Quiz questions

Given below is pseudocode for printing all the vertices of a connected undirected graph G.

```
mark all vertices of G as "unmarked";
Stack fringe = new Stack ( );
fringe.push (some vertex);
mark the vertex just pushed as "on the stack";
while (!fringe.isEmpty ( )) {
   Vertex u = fringe.pop ( );
   System.out.println (u);
   mark u as "visited";
   for each neighbor w of u that's marked "unmarked",
      push w onto fringe;
      mark w as "on the stack";
}
```

Suppose that G has N vertices and E edges. A CS 61B student claims that, when G is represented as an array of adjacency lists, this algorithm runs in time proportional to N^2. The student reasons that the outer loop executes N times, and a vertex may have N-1 neighbors.

1. Describe a connected undirected graph for which this estimate is significantly too large, and explain how long the algorithm takes for this graph.

2. Give a more accurate estimate of the algorithm's running time for any graph G, in terms of N, the number of vertices in G, and E, the number of edges in G. Briefly explain your answer.