Sorting

Sorting

- Fundamental process in programming and widely used.
- Sorting is the process of arranging a list of items into a particular order
- There are many algorithms for sorting a list of items
- These algorithms vary in efficiency
- Many practical applications in computing require things to be in order
  - Querying a database and appending an ORDER BY clause in fact performs sorting
  - A phone book directory is a list that has already been sorted (and imagine if it weren't!)

Sorting

- General Considerations
- Sort a deck of cards in ascending order
- Different approaches
  - Start making piles
  - Spread the cards all over the table
  - Juggle cards around their hands
- Takes seconds for some, for others several minutes or more
- Some sorted deck of cards will be arranged such that spades always appear before hearts
- In other cases, it might be less organized

Sorting

- There are many algorithms for sorting a list of items
- These algorithms vary in efficiency
- We will examine some algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
- Almost all modern languages provide built-in sort functions that can often be used when a sort is necessary

Bubble Sort

- One of the oldest sorts known to man due to its simplicity
- Based on the properly of a sorted list that any two adjacent elements are in sorted order
- In a typical iteration, each adjacent pair of elements is compared, starting with the first two elements, all the way to the final two elements
- Each time two elements are compared
  - if they are already in sorted order, nothing is done to them and the next pair of elements is compared
  - in the case where the two elements are not in sorted order, the two elements are swapped, putting them in order

Bubble Sort

- Let’s consider a set of data: 4 8 1 7 3 5 2.
- During first iteration of Bubble Sort:
- (4 8) 1 7 3 5 2 --> compare 4 and 8, no swap
- 4 (8 1) 7 3 5 2 --> compare 8 and 1, swap
- 4 1 (8 7) 3 5 2 --> compare 8 and 7, swap
- 4 1 7 (8 3) 5 2 --> compare 8 and 3, swap
- 4 1 7 3 (8 5) 2 --> compare 8 and 5, swap
- 4 1 7 3 5 (8 2) --> compare 8 and 2, swap
- 4 1 7 3 5 2 8 --> first iteration complete

Bubble Sort

- During the second iteration, the second-biggest element will be moved to the penultimate position:

  (4 1) 7 3 5 2 8 --> compare 4 and 1, swap

  1 (4 7) 3 5 2 8 --> compare 4 and 7, no swap

  1 4 (7 3) 5 2 8 --> compare 7 and 3, swap

  1 4 3 (7 5) 2 8 --> compare 7 and 5, swap

  1 4 3 5 (7 2) 8 --> compare 7 and 2, swap

  1 4 3 5 2 7 8 --> no need to compare last two

- In the next pass, 5 will be moved up to its position (third from last and so on

Bubble Sort

- Typical implementation

```
for (int i = 0; i < data.length; i++) {
        for (int j = 0; j < data.length - 1; j++){
                if (data[j] > data[j + 1]) {
                    tmp = data[j];
                    data[j] = data[j + 1];
                    data[j + 1] = tmp;
                }
        }
}
```

- $O(n^2)$ runtime, and hence is very slow for large data sets.
- Memory space complexity of $O(n)$
- However, very simple to understand and code
- Also, it is a stable sort that requires no additional memory, since all swaps are made in place

Bubble Sort

- Best case is when list is already sorted, then no swaps are made.
- If at any time a pass is made through the list, no swaps were made, it is certain that the list is sorted
- The absolute worst case is when the smallest element of the list is at the end
- It will not get to the front of the list until all n iterations have occurred. In this worst case, it takes of the order of n^2
- Best Case: n Average Case: n^2 Worst Case: n^2

Selection Sort

- The approach of Selection Sort:
  - find the smallest element and put it at the beginning of the list
  - repeat this process on the unsorted remainder of the data
- (One implementation) in more detail:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are placed
- Rather than making successive swaps with adjacent elements like Bubble Sort, Selection sort makes only one, swapping the smallest number with the number occupying its correct position

Selection Sort

Example: ***Sort 3, 9, 6 , 1, 2 using Selection sort***

```
  original:           3   9   6   1   2
  scan from 3 to 2, smallest is 1
  swap 3 and 1:       1   9   6   3   2
  scan from 9 to 2, smallest is 2
  swap 9 and 2:       1   2   6   3   9
  scan from 6 to 9, smallest is 3
  swap 6 and 3:       1   2   3   6   9
  scan from 6 to 9, smallest is 6
  swap 6 and 6 :      1   2   3   6   9  (redundant)
  scan from 9 to 9, smallest is 9
  swap 9 and 9 :      1   2   3   6   9  (redundant)
sorted…
```

Selection Sort

```
// sort by selection
public static void main (String[] argv)
   {
   int locofmin, temp, i, j;
   int[] ages = {3,   9,   6,   1,   2};
   for (int i = 0; i < ages.length-1; i++)
      {
      locofmin = i;
      for (j = i+1; j < ages.length; j++)
         {
         if (ages[j] < ages[locofmin])
            locofmin = j;
         }
      // Swap the values
      temp = ages[locofmin];
      ages[locofmin] = ages[i];
      ages[i] = temp;
      }
   }
```

Selection Sort

- Implemented with one loop nested inside another (like Bubble sort)
- Efficiency is n^2
  - First iteration through the data requires n - 1 comparisons to find the minimum value to swap into the first position.
  - First position can then be ignored when finding the next smallest value, the second iteration requires n - 2 comparisons and third requires n - 3 ...
  - progression is $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1)/2 = O(n^2)$
- Efficiency independent of data (unlike Bubble sort)

  e.g. it cannot recognise (on the 1st iteration) the difference between the following two sets of data: 1 2 3 4 5 6 7 8 9 and 1 9 8 7 6 5 4 3 2.
- In each case, it will identify the 1 as the smallest element and then go on to sorting the rest of the list.
- there is no best or worst cases since it treats all data sets the same so efficiency is always $O(n^2)$

Insertion Sort

- The approach of Insertion Sort:
  - Pick any item and insert it into its proper place in a sorted sublist
  - repeat until all items have been inserted

- (One implementation) in more detail:
  - consider the first item to be a sorted sublist (of one item)
  - insert the second item into the sorted sublist, shifting items as necessary to make room to insert the new addition
  - insert the third item into the sorted sublist (of two items), shifting as necessary
  - repeat until all values are inserted into their proper position

Insertion Sort

Example: ***Sort 3, 9, 6 , 1, 2 using Insertion sort***

original: 3 9 6 1 2

start from 9, as 9>3 insert 9 in position 1 (redundant)

after inserting 9: 3 9 6 1 2

start from 6, as 6<9 move 9 to position 2, as 6>3 insert 6 in position 1

after inserting 6: 3 6 9 1 2

start from 1, as 1<9 move 9 to position 3, as 1<6 move 6 to position 2, as 1<3 move 3 to position 1, insert 1 in position 0

after inserting 1: 1 3 6 9 2

start from 2, as 2<9 move 9 to position 4, as 2<6 move 6 to position 3, as 2<3 move 3 to position 2, as 2>1 insert 2 in position 1

after inserting 2: 1 2 3 6 9

sorted…

Insertion Sort

```
public static void main (String[] argv)
{
int key, i, pos;
int[] ages = {3,   9,   6,   1,   2};
for (i = 1; i < ages.length; i++)
   {
      key = ages[i];
      pos = i;
      // shift larger values to the right
      while (pos > 0 && ages[pos-1]  > key)
      {
         ages[pos] = ages[pos-1];
         pos=pos-1;
      }
      ages[pos] = key;
   }
}
```

Insertion Sort

- To determine the average efficiency of insertion sort consider the number of times that the inner loop iterates
- As with other nested loop features, the number of iterations follows a familiar pattern $1 + 2 + ... + (n - 2) + (n - 1) = n(n - 1)/2 = O(n^2)$
  - i.e. it takes one iteration to build a sorted sublist of length 1, 2 iterations to build a sorted sublist of length two ... n-1 iterations to build the final list
- The best case is on a sorted list where it runs $O(n)$
  - Where data is already sorted, insertion sort won't have to do any shifting because the local sublist will already be sorted
  - i.e. the first element will already be sorted, the first two will already be sorted, the first three, and so on
  - thus, insertion sort will iterate once through the list, and, finding no elements out of order, will not shift any of the data around

Quicksort

- Fast sorting algorithm
- Compares items and swaps them if they are out of sequence
- However, here the list is divided into smaller lists which can then be sorted (*Divide and Conquer* approach)
- Usually done through **recursion**
- The premise of quicksort is to separate the "big" elements from the "small" elements repeatedly

Quicksort

- Choose a "pivot" value that will be used to divide big and small numbers
  - different methods to choose a pivot value
  - e.g. we can simply use the first element of the list as the pivot value
- Once the pivot value selected
  - all values smaller than the pivot are placed toward the beginning of the set
  - all the ones larger than the pivot are placed to the right
  - this process essentially sets the pivot value in the correct place each time
- Apply quicksort algorithm recursively to the left and the right parts (each side of the pivot)

Quicksort

- Let's consider the data set: 1 12 5 26 7 14 3 7 2

1 12 5 26 7 14 3 7 2 unsorted

1 12 5 26 **7** 14 3 7 2 pivot value = 7

1 **12** 5 26 **7** 14 3 7 **2** 1<7 (ignore) 12 > 7 and 7 >= 2, swap 12 and 2

1 2 5 **26** **7** 14 3 **7** 12 5<7 (ignore) 26 > 7 and 7 >= 7, swap 26 and 7

1 2 5 7 **7** **14** **3** 26 12 14 => 7 and 7 >= 3, swap 14 and 3

1 2 5 7 **7** 3 14 26 12 **left and right pointers cross so swap pivot with element before left pointer**

1 2 5 7 3 **7** 14 26 12 run quicksort recursively

…

1 2 3 5 7 7 12 14 26 sorted

Quicksort

- Efficiency is entirely dependent upon the “pivot” value
- Ideally, the pivot would be selected such that it were smaller than about half the elements and larger than about half the elements
- If the data to be sorted is known to all fit within a given range, or fit a certain distribution model, this knowledge can be used to improve the efficiency by choosing pivot values that are likely to divide the data in half as close to evenly as possible
- In a best case the runtime is $O(n * \log n)$.
- In the worst case the runtime drops to $O(n^2)$
- The proof for efficiency is not trivial and not presented here

Comparing Sorts

- Three things to consider when comparing sorting algorithms

Runtime :

- When dealing with increasingly large sets of data, inefficient sorting algorithms can become too slow for practical use within an application.

Memory space :

- Faster algorithms that required recursive calls typically involve creating copies of the data to be sorted
- In some environments where memory space may be at a premium (such as in an embedded system) certain algorithms may be impractical

Stability :

- This is what happens to elements that are comparatively the same
- In a stable sort, those elements whose comparison key is the same will remain in the same relative order after sorting as they were before sorting

Comparing Sorts

Algorithm	Order	Implementation Complexity	Notes
Bubble Sort	$O(n^2)$	Very low	In place Slow (good for small inputs, <1K)
Selection Sort	$O(n^2)$	Very low	In place Slow (good for small inputs, <1K)
Insertion Sort	$O(n^2)$	Very low	In place Slow (good for small inputs, <1K)
Quick Sort	$O(n \log n)$	High	In place, randomised Fastest (good for large inputs, >1M)

References

- Just to show how many sorting algorithms there are…
- "Sorting" by William A. Martin, in ACM Computing Surveys, Volume 3, Number 4, December 1971, pages 147-174.
- From the abstract for this paper:

"*The bibliography appearing at the end of this article lists 37 sorting algorithms and 100 books and papers on sorting published in the last 20 years. The ideas presented here have been abstracted from this body of work, and the best algorithms known are given as examples. As the algorithms are explained, references to related algorithms and mathematical or experimental analyses are given. Suggestions are then made for choosing the algorithm best suited to a given situation.*"

... and that was over 40 years ago!