CSE 373

Sorting 2: Selection, Insertion, Shell Sort

reading: Weiss Ch. 7

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Selection sort

- **selection sort**: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

  The algorithm:
  - Look through the list to find the smallest value.
  - Swap it so that it is at index 0.

  - Look through the list to find the second-smallest value.
  - Swap it so that it is at index 1.

    ...

  - Repeat until all values are in their proper places.

Selection sort example

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	**-4**	18	12	**22**	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	**2**	12	22	27	30	36	50	7	68	91	56	**18**	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	**7**	22	27	30	36	50	**12**	68	91	56	18	85	42	98	25

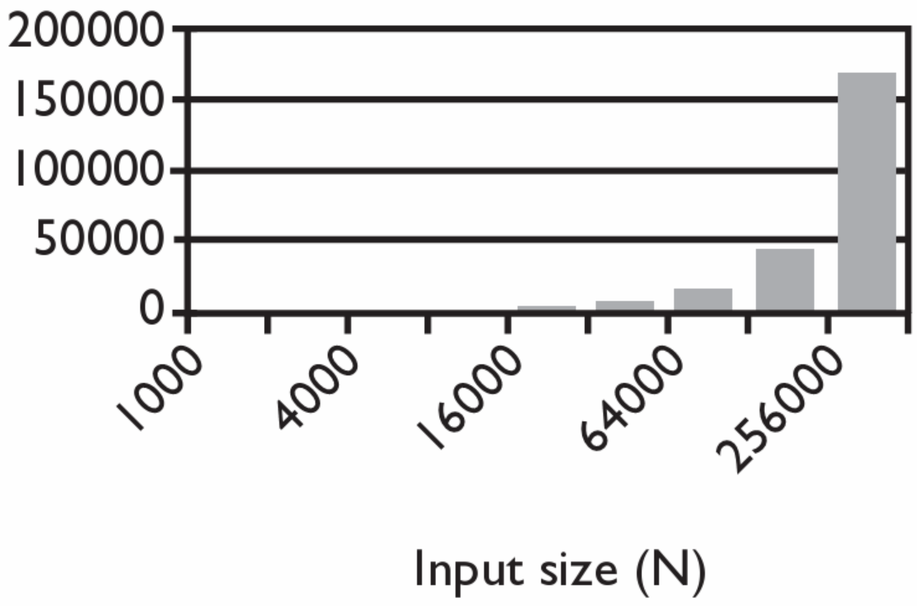
Selection sort code

```java
// Rearranges the elements of a into sorted order using
// the selection sort algorithm.
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // swap smallest value to its proper place, a[i]
        swap(a, i, min);
    }
}
```

Selection sort runtime

- What is the complexity class (Big-Oh) of selection sort?

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Selection sort runtime

- Running time for input size *N*:
  - in practice, a bit faster than bubble sort. Why?

$$\sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1} 1 = \sum_{i=0}^{N-1}\left(N-1-(i+1)+1\right)$$

$$= \sum_{i=0}^{N-1}\left(N-i+1\right)$$

$$= N\sum_{i=0}^{N-1} 1 - \sum_{i=0}^{N-1} i$$

$$= N^2 - N - \frac{(N-1)N}{2}$$

$$= O(N^2)$$

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert second item into sorted sublist, shifting first item if needed
  - insert third item into sorted sublist, shifting items 1-2 as needed
  - ...
  - repeat until all values have been inserted into their proper positions

Insertion sort example

- Makes *N*-1 passes over the array.
- At the end of pass *i*, the elements that occupied *A*[0]…*A*[*i*] originally are still in those spots and in sorted order.

index	0	1	2	3	4	5	6	7
value	15	2	8	1	17	10	12	5
pass 1	**2**	**15**	8	1	17	10	12	5
pass 2	**2**	**8**	**15**	1	17	10	12	5
pass 3	**1**	**2**	**8**	**15**	17	10	12	5
pass 4	**1**	**2**	**8**	**15**	**17**	10	12	5
pass 5	**1**	**2**	**8**	**10**	**15**	**17**	12	5
pass 6	**1**	**2**	**8**	**10**	**12**	**15**	**17**	5
pass 7	**1**	**2**	**5**	**8**	**10**	**12**	**15**	**17**

Insertion sort code

```java
// Rearranges the elements of a into sorted order.
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements right to make room for a[i]
        int j = i;
        while (j >= 1 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Insertion sort runtime

- *worst case:* reverse-ordered elements in array.

$$\sum_{i=1}^{N-1} i = 1 + 2 + 3 + \ldots + (N-1) = \frac{(N-1)N}{2}$$

$$= O(N^2)$$

- *best case:* array is in sorted ascending order.

$$\sum_{i=1}^{N-1} 1 = N - 1 = O(N)$$

- *average case:* each element is about halfway in order.

$$\sum_{i=1}^{N-1} \frac{i}{2} = \frac{1}{2}(1 + 2 + 3 \ldots + (N-1)) = \frac{(N-1)N}{4}$$

$$= O(N^2)$$

Inversions and average case

- **inversion**: Given an array A of elements, an inversion is an ordered pair (i, j) such that $i < j$, but $A[i] > A[j]$. *(out of order elements)*
  - Assume no duplicate elements.

- *Theorem:* The average number of inversions in an array of N distinct elements is $N (N - 1) / 4$.
  - *Corollary:* Any algorithm that sorts by exchanging adjacent elements requires $O(N^2)$ time on average.

Shell sort

- **shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959

- based on some observations about insertion sort:
  - insertion sort runs fast if the input is almost sorted
  - insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position

  - Runs a lot faster on already-sorted ascending input than random input (no shifting needed). Also works well on descending input.

Shell sort example

- For some sequence of gaps $g_1, g_2, g_3, \ldots, 1$:
  - Sort all elements that are g_1 indexes apart *(using insertion sort)*
  - Then sort all elements that are g_2 indexes apart, ...
  - Then sort all elements that are 1 index apart *(using insertion sort)*
- An example that sorts by gaps of 8, then 4, then 2, then 1:

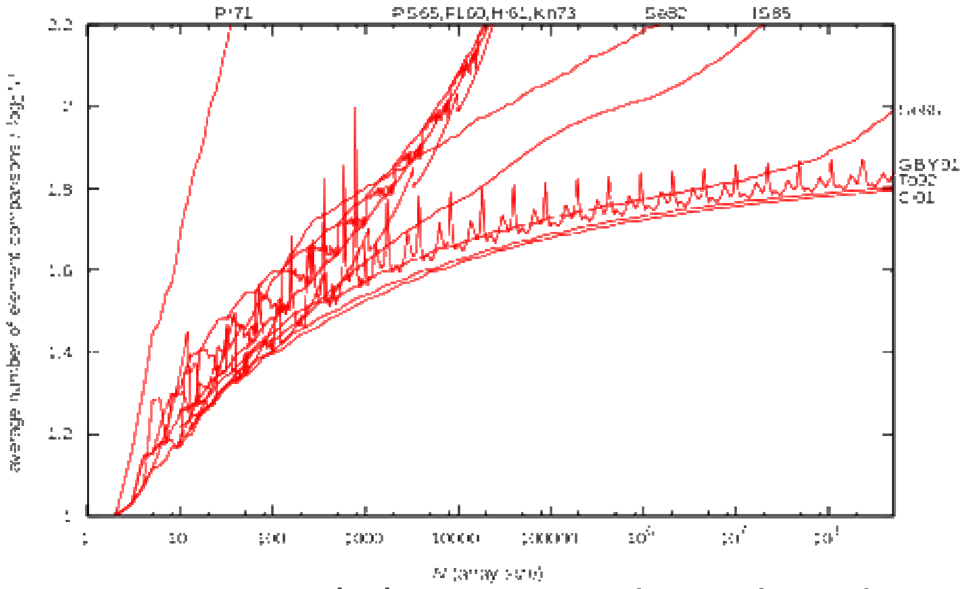
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
start	27	88	92	-4	22	30	36	50	7	18	11	76	2	65	56	3	85
gap 8	7	18	11	-4	2	30	36	3	27	88	92	76	22	65	56	50	85
gap 4	2	18	11	-4	7	30	36	3	22	65	56	50	27	88	92	76	85
gap 2	2	-4	7	3	11	18	22	30	27	50	36	65	56	76	85	88	92
gap 1	-4	2	3	7	11	18	22	27	30	36	50	56	65	76	85	88	92

Shell sort code

```
// Rearranges the elements of a into sorted order.
// Uses a shell sort with gaps that divide by 2:
// length/2, length/4, ..., 4, 2, 1
public static void shellSort(int[] a) {
    for (int gap = a.length / 2; gap >= 1; gap /= 2) {
        for (int i = gap; i < a.length; i++) {
            // slide elements right by 'gap'
            // to make room for a[i]
            int temp = a[i];
            int j = i;
            while (j >= gap && a[j - gap] > temp) {
                a[j] = a[j - gap];
                j -= gap;
            }
            a[j] = temp;
        }
    }
}
```

Shell sort runtime

- More difficult to analyze than runtime of previous sorts.



  - ~ O($N^{1.25}$) on average; O(N) on ascending already-sorted input
  - (Wikipedia) Applying the theory of Kolmogorov complexity, Jiang, Li, and Vitányi proved the following lower bounds for the order of the average number of operations in an m-pass Shellsort: O($mN^{1+1/m}$) when $m=\log_2 N$ and O(mN) when $m>\log_2 N$. Therefore Shellsort has prospects of running in an average time that asymptotically grows like NlogN only when using gap sequences whose number of gaps grows in proportion to the logarithm of the array size. It is, however, unknown whether Shellsort can reach this asymptotic order of average-case complexity, which is optimal for comparison sorts.