Simple Sorting Algorithms

EECS 233

Basics of Sorting

- Array data structure
- Ground rules:
  - sort the values in increasing order
  - sort “in place”, using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of **comparisons *C*** and the number of **moves *M*** needed to sort the array.
  - comparison = compare the keys of two elements
  - move = copying an element from one position to another

Defining Methods for Sorting

- In Java, we can put them in a Sort class that is simply a collection of methods

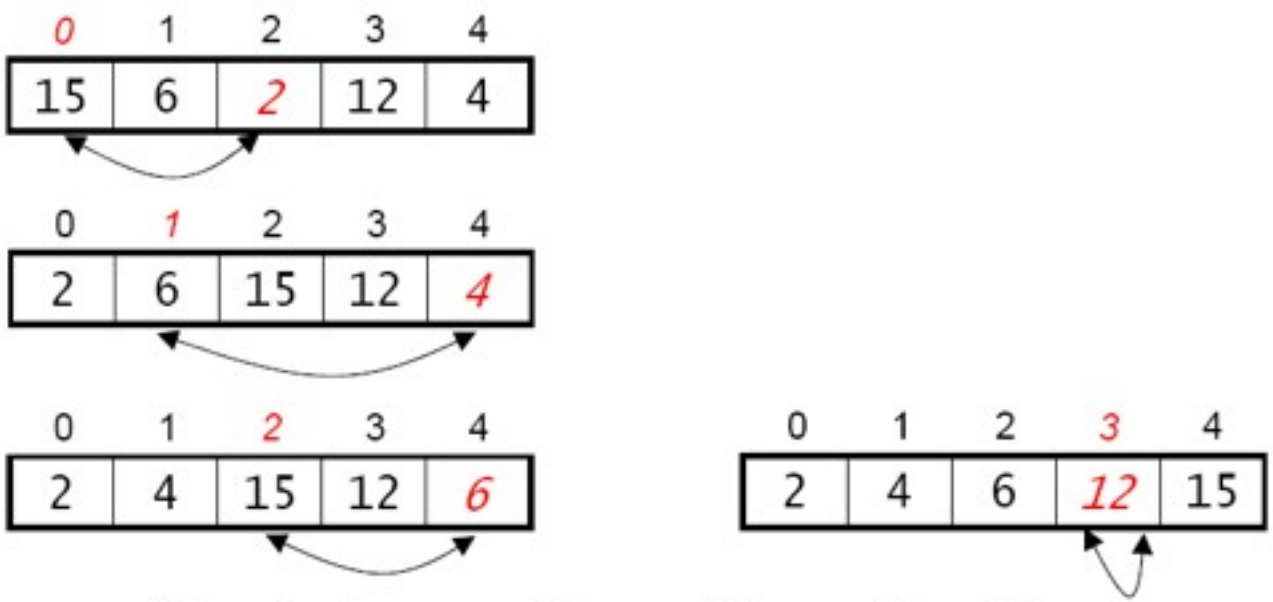
```
public class Sort {
        static void bubbleSort(int[] arr) {
                ...
        }
        static void insertionSort(int[] arr) {
                ...
        }
        ...
}
```

- We never create Sort objects. All of the methods in the class must be *static*
- Outside the class, we invoke them using the class name:
  - e.g., Sort.bubbleSort(arr);

Method 1: Selection Sort

- Basic idea:
  - consider the positions in the array from left to right
  - for each position, find the element that belongs there and put it in place by swapping it with the element that's currently there

- An example:



Selecting An Element

- When we consider position i, the elements in positions 0 through i – 1 are already in their final positions.

0	1	2	3	4	5	6
2	4	7	21	25	10	17

- To select an element for position i,
  - consider elements i, i+1,i+2,…,arr.length – 1, and keep track of indexMin, the index of the smallest element seen thus far

0	1	2	3	4	5	6
2	4	7	21	25	*10*	17

  - when we finish this pass, indexMin is the index of the element that belongs in position i.
  - swap arr[i] and arr[indexMin]:

0	1	2	3	4	5	6
2	4	7	10	25	21	17

Implementation of Selection Sort

- The sort method is very simple:

```
static void selectionSort(int[] arr, int length) {
    for (int i = 0; i < length – 1; i++) {
        int j = indexSmallest(arr, i, length – 1);
        swap(arr, i, j);
    }
}
```

- It uses a helper method to find the index of the smallest element:

```
static int indexSmallest(int[] arr, int lower, int upper) {
    int indexMin = lower;
    for (int i = lower+1; i <= upper; i++)
        if (arr[i] < arr[indexMin])
            indexMin = i;
    return indexMin;
}
```

0	1	2	3	4	5	6
2	4	7	21	25	10	17

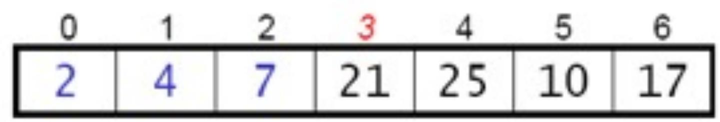
Running Time Analysis

- Input size n: the # of elements in the array
- Time metrics:
  - C(n) = number of comparisons
  - M(n) = number of moves

Number of Comparisons

- To sort n elements, selection sort performs n - 1 passes:
  - on 1st pass, it performs n - 1 comparisons to find indexSmallest
  - on 2nd pass, it performs n - 2 comparisons
  - …
  - on the (n-1)st pass, it performs 1 comparison

```
static void selectionSort(int[] arr, int length) {
        for (int i = 0; i < length – 1; i++) {
                int j = indexSmallest(arr, i, length – 1);
                swap(arr, i, j);
        }
}
static int indexSmallest(int[] arr, int lower, int upper) {
        int indexMin = lower;
        for (int i = lower+1; i <= upper; i++)
                if (arr[i] < arr[indexMin])
                        indexMin = i;
        return indexMin;
}
```



- Adding up the comparisons, $C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1) = n^2/2 – n/2$

Number of Moves

- Moves: after each of the n-1 passes to find the smallest remaining element, the algorithm *may* perform a swap to put the element in place.

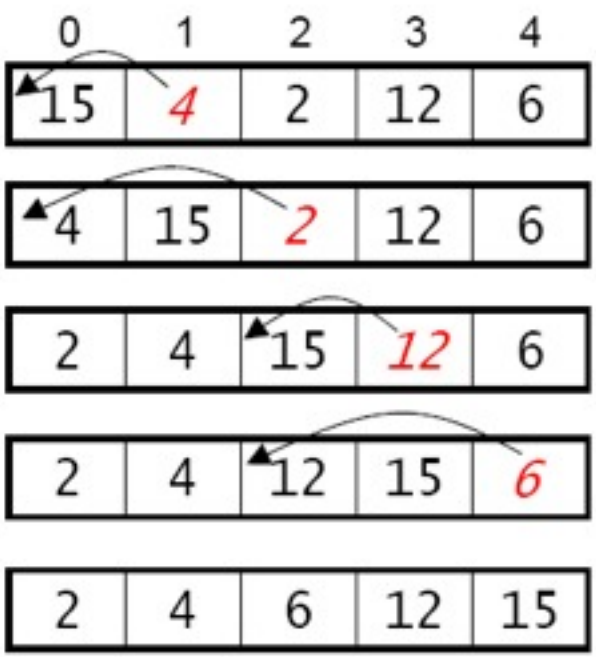
- At most n–1 swaps, 3 moves per swap
  - $M(n) = 3(n-1) = 3n-3$
  - selection sort performs $O(n)$ moves.

- Considering both comparisons and moves, the overall running time is $O(n^2)$

Method 2: Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sliding over” other elements to make room.

- An example:



Distinguishing Selection Sort and Insertion Sort

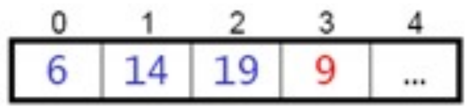
- Selection sort: loop through positions in the array and select the correct elements from the subsequent array to fill them
- Insertion sort: loop through elements and determine where to insert them in the preceding array.

0	1	2	3	4	5	6	7	8
16	8	13	2	15	9	4	12	24

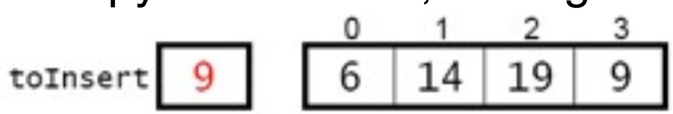
- An example that illustrates the difference:
  - Sorting by selection:
    - consider position 0: find the element (“2”) that belongs there
    - consider position 1: find the element (“4”) that belongs there
    - …
  - Sorting by insertion:
    - consider element “8”: determine where to insert it
    - consider element “13”; determine where to insert it
    - …

Inserting An Element

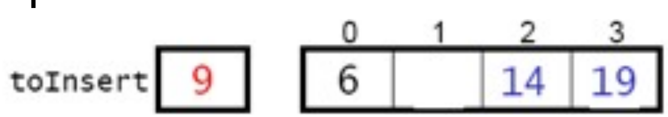
- When we consider element i, elements 0 through i – 1 are already sorted with respect to each other.
  - i = 3:



- To insert element i:
  - make a copy of element i, storing it in the variable toInsert:



  - consider elements i-1, i-2, …
    - ✓ if an element > toInsert, slide it over to the right
    - ✓ stop at the first element <= toInsert

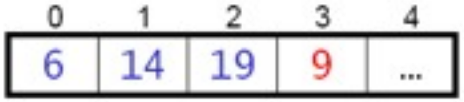


  - copy toInsert into the resulting “hole”:

Implementation of Insertion Sort

0	1	2	3	4	5	6	7	8
16	4	15	7	8	10	2	3	5

```
static void insertionSort(int[] arr, int length) {
        for (int i = 1; i < length; i++) {
                if (arr[i] < arr[i-1]) {
                        int toInsert = arr[i];
                        int j = i;
                        while (j > 0 && toInsert < arr[j-1]) {
                                arr[j] = arr[j-1];
                                j = j - 1;
                        }
                        arr[j] = toInsert;
                }
        }
}
```



Running time Analysis

- The number of operations depends on the contents of the array.
  - *best case:*
    - array is sorted
    - thus, we never execute the do-while loop
    - each element is only compared to the element to its left
    - $C(n) = n - 1 = O(n)$, $M(n) = 0$, running time $= O(n)$
  - *worst case:*
    - array is in reverse order
    - each element is compared to *all* of the elements to its left:
      - arr[1] is compared to 1 element (arr[0])
      - arr[2] is compared to 2 elements (arr[0] and arr[1])
      - …
      - arr[n-1] is compared to n-1 elements
      - $C(n) = 1 + 2 + \ldots + (n - 1) = O(n^2)$ as seen in selection sort
      - similarly, $M(n) = O(n^2)$, running time $= O(n^2)$
  - *average case:*
    - elements are randomly arranged
    - each element is compared to *half* of the elements to its left
    - still get $C(n) = M(n) = O(n^2)$, running time $= O(n^2)$

An Improvement?

```
static void insertionSort(int[] arr, int length) {
        for (int i = 1; i < length; i++) {
                if (arr[i] < arr[i-1]) {
                        int toInsert = arr[i];
                        int j = i;
                        while (j > 0 && toInsert < arr[j-1]) {
                                arr[j] = arr[j-1];
                                j = j - 1;
                        }
                        arr[j] = toInsert;
                }
        }
}
```

- The array to the left of the current element is already sorted.
  - Use binary search to find the proper position to insert toInsert!
  - Would be log(n) comparisons per element.
- Then what would be the running times in best/worst/average cases?

Selection Sort or Insertion Sort?

- For sorted or nearly sorted arrays, insertion sort is *much* faster.
  - insertion sort = $O(n)$
  - selection sort = $O(n^2)$

- For random data, they are roughly equivalent (both $O(n^2)$)
  - selection sort requires more comparisons
    - selection = $n^2/2 - n/2$ *always*
    - insertion = $n^2/4 - n/4$ in the avg case
      - when insertion enters the loop, it stops once arr[j] >= toCompare
  - insertion sort requires *much* more moves
    - insertion = $O(n^2)$
    - selection = $O(n)$

- For an array in reverse order, selection sort is faster.
  - why?

Method 3: Shell sort

- Developed by Donald Shell in 1959

- Improves on insertion sort, and takes advantage of the fact that insertion sort is fast when an array is almost sorted.

- Also seeks to eliminate a disadvantage of insertion sort:
  - if an element is far from its final location, many “small” moves are required to put it where it belongs.
  - Example: if the largest element starts out at the beginning of the array, it moves one place to the right on *every* insertion!

0	1	2	3	4	5	6	7	8
99	**8**	**13**	**2**	**15**	**9**	**4**	**12**	**24**

  - Shell sort uses “larger” moves that allow elements to quickly get close to where they belong.

Shell Sort: Basic Ideas

- Sorting Subarrays
  - use insertion sort on interleaved subarrays that contain elements separated by some increment
  - larger increments allow the data items to make quicker “jumps”
  - repeatedly using a decreasing sequence of increments

- Example for an initial increment of 3 (3 subarrays)

0	1	2	3	4	5	6	7	8
99	8	13	2	15	9	4	12	24

- Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7	8
2	8	9	4	12	13	99	15	24

- Finally, we complete the process using an increment of 1.

Single-Pass Shell Sort

- We *don't* consider the subarrays one at a time.
- We consider elements arr[incr] through arr[arr.length-1], inserting each element into its proper place with respect to the elements *from its subarray* that are to the left of the element.
- Example (increment = 3):

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8
27	18	10	36	3	20	9	8
27	3	10	36	18	20	9	8
27	3	10	36	18	20	9	8
9	3	10	27	18	20	36	8
9	3	10	27	8	20	36	18

Choosing The Sequence of Increments

- Different sequences of decreasing increments can be used.
  - The last increment should be 1. A 1-sorted array is sorted; a 3-sorted array is only partially sorted.
- A good sequence (Hibbard's): one less than a power of two.
  - $2^k - 1$ for some k: … 63, 31, 15, 7, 3, 1
  - can get to the next lower increment using integer division:

    incr = incr/2;
- The sequence of increments should avoid numbers that are multiples of each other.
  - A bad sequence: … 64, 32, 16, 8, 4, 2, 1

Implementation of Shell Sort

```
static void shellSort(int[] arr, int length) {
        int incr = 1;
        while (2 * incr <= length) incr = 2 * incr;
        incr = incr - 1;

        while (incr >= 1) {
                for (int i = incr; i < length; i++) {
                        if (arr[i] < arr[i-incr]) {
                                int toInsert = arr[i];
                                int j = i;
                                while (j > incr-1  && toInsert < arr[j-incr]) {
                                    arr[j] = arr[j-incr];
                                    j = j - incr;
                                }
                                arr[j] = toInsert;
                        }
                }
                incr = incr/2;
        }
}
```

- The highlighted code is from insertionSort() except that incr replaces 1

Running Time of Shell Sort

- Depends on the sequence of decreasing increments
  - Should decrease fast to lower the number of passes of insertion sort
  - Should not decrease too fast to be close to (one-pass) insertion sort
- Hibbard's sequence has worst-cast running time at $O(n^{3/2})$; similar to
- (Case Alum) Knuth's sequence: … 1093, 364, 121, 40, 13, 4, 1

  incr = incr/3;
- Typical approach: the sequence decreases exponentially, meanwhile a pair of increments should try to be prime to each other
- The running time (worst-case and best-case) is often difficult to analyze, and some bounds are unknown.
  - What is the average-case running time for Hibbard sequence?