SORTING

MSCI 240: Algorithms & Data Structures

lecture summary

sorting overview

selection sort

insertion sort

complexity of selection & insertion

Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity	chapter 13	1.4
Searching		pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
Binary Search Trees	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

sorting: rearranging the values in an array or collection into a specific order (usually into their “natural ordering”)

- one of the fundamental problems in computer science

can be solved in many ways:

- many sorting algorithms
- some are faster/slower than others
- some use more/less memory than others
- some work better with specific kinds of data
- some can utilize multiple computers / processors, ...

the `Arrays` and `Collections` classes in `java.util` have a static method `sort` that sorts the elements of an array/list

```
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]

ArrayList<String> words2 = new ArrayList<>();
for (String word : words) {
    words2.add(word);
}
Collections.sort(words2);
System.out.println(words2);
// [ball, bar, baz, foo]
```

`Collections` class

Method name	**Description**
`binarySearch(`**list**`, `**value**`)`	returns the index of the given value in a sorted list (< 0 if not found)
`copy(`**listTo**`, `**listFrom**`)`	copies **listFrom**'s elements to **listTo**
`emptyList(), emptyMap(), emptySet()`	returns a read-only collection of the given type that has no elements
`fill(`**list**`, `**value**`)`	sets every element in the list to have the given value
`max(`**collection**`), min(`**collection**`)`	returns largest/smallest element
`replaceAll(`**list**`, `**old**`, `**new**`)`	replaces an element value with another
`reverse(`**list**`)`	reverses the order of a list's elements
`shuffle(`**list**`)`	arranges elements into a random order
`sort(`**list**`)`	arranges elements into ascending order

sorting

input:

sequence of n numbers, $A = < a_1, a_2, \dots, a_n >$

output:

reordering (permutation) A' of A

where $A' = < a'_1, a'_2, \dots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

important algorithm properties

comparison-based

determine order by comparing pairs of elements: <,>,`compareTo`

stable

elements of the same value stay in the same order

e.g., sorting email by date, then by sender

in-place

constant amount of extra storage

sorting algorithms:

- bubble sort: swap adjacent pairs that are out of order
- selection sort: look for the smallest element, move to front
- insertion sort: build an increasingly large sorted front portion
- merge sort: recursively divide the array in half and sort it
- heap sort: place the values into a sorted tree structure
- quick sort: recursively partition array based on a middle value

other specialized sorting algorithms:

- bucket sort: cluster elements into smaller groups, sort them
- radix sort: sort integers by last digit, then 2nd to last, then ...
- ...

algorithm	worst-case	average	stable	in-place
selection sort	$O(n^2)$	$O(n^2)$	no	yes
insertion sort	$O(n^2)$	$O(n^2)$	yes	yes
mergesort	$O(n \log n)$	$O(n \log n)$	yes*	no
quicksort	$O(n^2)$	$O(n \log n)$	no	yes

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

activity: selection sort

select the next smallest element each time

selection sort example:

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

```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);
    }
}
```

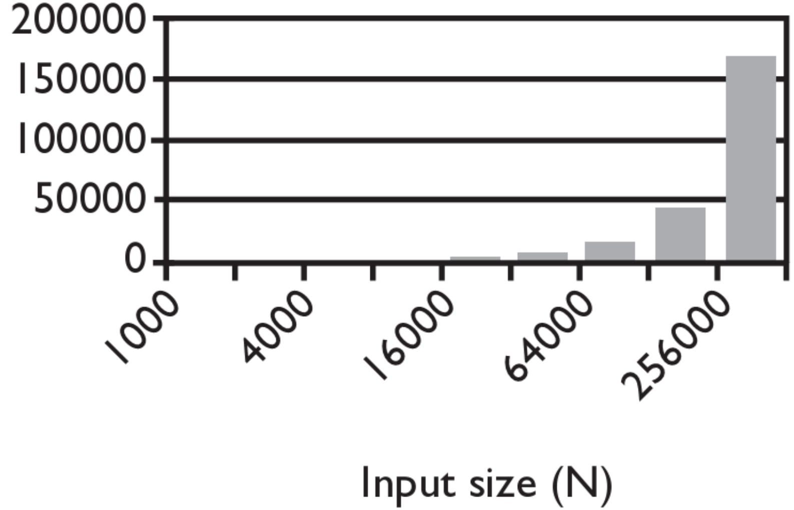
```java
// Swaps a[i] with a[j].
public static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

complexity of swap?

$T_{sw}(n) = 3$

what is the complexity class (big-oh) of selection sort?
(fig. 13.6)

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



```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);
    }
}
```

$(n-1) \rightarrow$ `int min = i;`

`if (a[j] < a[min]) {` $\leftarrow (n-1) + (n-2) + \cdots + 1$

`min = j;` $\leftarrow \frac{n(n-1)}{2}$

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$3(n-1) \rightarrow$ `swap(a, i, min);`

$$\boxed{T(n) = n^2 + 3n - 4 \ \in O(n^2)}$$

insertion sort: orders a list of values by shifting each element into a sorted sub-array

the algorithm:

- insert index `0` into sorted subarray of size `1` (already sorted)
- insert index `1` into sorted subarray of size `2` (shift left to insertion point)
- insert index `2` into sorted subarray of size `3` (shift left to insertion point)
  - …
- insert index n`-1` into sorted subarray of size n

activity: insertion sort

insert the next element in the “right” spot

```java
// Rearranges the elements of a into sorted order
// using the selection sort algorithm.
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        // shift element i left
        // until it's in the right spot
        int j = i;
        while (j > 0 && a[j] < a[j - 1]) {
            swap(a, j, j - 1);
            j--;
        }
    }
}
```

insertion sort example:

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

1st pass (size 2):

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

2nd pass (size 3):

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

insertion sort example:

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

3rd pass (size 4):

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

insertion sort example:

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

4th pass (size 5):

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

no change!

complexity of insertion sort

best case?

array already sorted (always leave in place—no need to shift left)

worst case?

array in reverse order (have to shift all the way to left every time)

best case

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
 (n − 1) → int j = i;
 (n − 1) → while (j > 0 && a[j] < a[j - 1]) {
            0 → swap(a, j, j - 1);
            0 → j--;
          }
    }
}
```

$$\boxed{T(n) = 2n - 2 \in O(n)}$$

worst case

```java
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int j = i;
        while (j > 0 && a[j] < a[j - 1]) {
            swap(a, j, j - 1);
            j--;
        }
    }
}
```

$(n-1) \rightarrow$ `int j = i;`

$$\left.\begin{array}{l}\texttt{swap(a, j, j - 1);}\\ \texttt{j--;}\end{array}\right\} 4\sum_{i=1}^{n-1} i = 4 \cdot \frac{n(n-1)}{2}$$

$$\boxed{T(n) = 2n^2 - n - 1 \in O(n^2)}$$

http://www.sorting-algorithms.com/

http://www.sorting-algorithms.com/selection-sort

http://www.sorting-algorithms.com/insertion-sort

summary

there are many sorting algorithms

properties: (comparison-based), stable, in-place

selection sort

select element that belongs in each index

insertion sort

insert next element into proper spot in sorted sub-array next:

mergesort