CSE 373 Section Handout #8
Sorting Algorithm Reference

bubble sort:

Repeatedly loop over the array, swapping neighboring elements if they are out of relative order.

index	0	1	2	3	4	5
value	42	77	35	12	91	8

42 77
35 77
12 77
77 91
8 91

value	42	35	12	77	8	91

- O(N^2) average, O(N) if input is sorted.
- Slow; performs lots of loops and swaps.

selection sort:

Repeatedly loop over the array, finding the smallest element, and swapping it to the front.

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

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

- O(N^2) in all cases
- Faster than bubble sort; makes N-1 swaps

insertion sort:

For increasing values of *i*, slide element *i* left until it is sorted with respect to elements [0 .. *i*-1].

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**

- O(N^2) average, O(N) if input is sorted.
- Faster than selection, especially on sorted data.

shell sort:

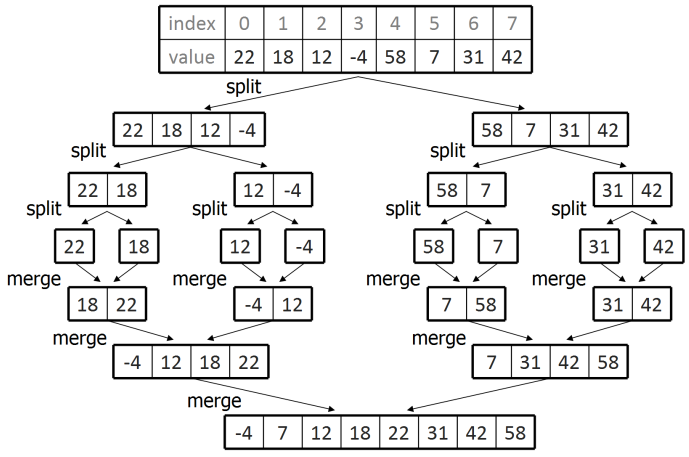
Perform an insertion sort on every *k*th element for decreasing values of *k* (e.g. *k*=8, 4, 2, 1)

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	92	50	27	88	56	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

- O($N^{1.25}$), O(N) if input is sorted.
- Generally faster than selection/insertion sort.

merge sort:

Split array in half, sort the halves, then merge the sorted halves back together.



- O(N log N) in all cases.
- A fast, general purpose, "stable" sort.

quick sort:

Choose some element as the "pivot". Partition the array into two groups: elements < pivot, and ≥ pivot. Then recursively repeat the process on each group.

index	0	1	2	3	4	5	6	7	8	9	
value	**65**	23	81	43	92	39	57	16	75	32	choose pivot=65
	32	23	81	43	92	39	57	16	75	**65**	swap pivot (65) to end
	32	23	**16**	43	92	39	57	**81**	75	65	swap 81, 16
	32	23	16	43	**57**	39	**92**	81	75	65	swap 57, 92
	32	23	16	43	57	39	92	81	75	65	
	32	23	16	57	43	39	**65**	81	75	**92**	swap pivot back in

recursively quicksort each half

32	23	16	57	43	39	pivot=32
39	23	16	57	43	**32**	swap to end
16	23	**39**	57	43	32	swap 39, 16
16	23	**32**	57	43	**39**	swap 32 back in

...

81	75	92	pivot=81
92	75	**81**	swap to end
75	92	81	swap 92, 75
75	**81**	**92**	swap 81 back in

...

- O(N log N) average, O(N^2) worst-case.
- extremely fast but not "stable"
- Choosing pivot poorly can hurt performance.

CSE 373 Section Handout #8

The problems on this page refer to the following arrays:

```
   index    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
a)        {29, 17,  3, 94, 46,  8, -4, 12}
b)        {14, 25, 95,  0, 17, -2, 13, 56, 34}
c)        { 6,  7,  4, 11,  8,  1, 10,  3,  5,  2}
d)        { 7,  1,  6, 12, 18,  8,  4, 21,  2, 30, -1,  9, -3, 10, 11, 27, 14}
```

1. **Bubble Sort Tracing**
   Trace the execution of the *bubble sort* algorithm over array a) above. Show each pass of the algorithm and the state of the array after the sweep has been performed, until the array is sorted.

2. **Selection Sort Tracing**
   Trace the execution of the *selection sort* algorithm over array b) above. Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted.

3. **Insertion Sort Tracing**
   Trace the execution of the *insertion sort* algorithm over array c) above. Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted.

4. **Shell Sort Tracing**
   Trace the execution of the *shell sort* algorithm over array d) above. Use gaps of *N*/2, *N*/4, ..., 2, 1. (In this case, that comes out to gaps of 8, 4, 2, then 1.) Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted.

For more practice later, try performing the algorithms over the other arrays and see the results.

CSE 373 Section Handout #8

5. `dualSelectionSort`

Write a method named `dualSelectionSort` that performs the selection sort algorithm on an array of integers. Your code should modify the algorithm shown in class by grabbing two elements on each pass through the array: the smallest and the largest, and move the smallest to the front and the largest to the end.

For example, if an array variable `a` stores the values `{58, 64, 1, 72, 63, 27, 9, 14}`, then making the call of `dualSelectionSort(a);` would make the following passes over `a` to sort it:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Value	58	64	1	72	63	27	9	14
Pass 0	**1**	64	58	14	63	27	9	**72**
Pass 1	1	**9**	58	14	63	27	**64**	72
Pass 2	1	9	**14**	58	27	**63**	64	72
Pass 3	1	9	14	**27**	**58**	63	64	72

(There is one tricky case to watch out for, which we will call the "*max=i*" case. Suppose your code has just finished pass #*i* and has index variables *min* and *max*. Suppose the maximum value is the first one, the value at index *i*. If you swap *min* with *i*, now when you swap *max* to the end, your code will mistakenly grab the minimum out of index *i* and move it to the end. You can see this above on Pass 1, where 64 is the max and it is in the first slot, index 1. The quick fix for this *max=i* case is to set *max=min* when *max=i*.)

As you write the method, consider the following questions:

- Do you expect your dual selection sort to be faster than a regular selection sort? Why or why not?
- What is the Big-Oh of your dual selection sort algorithm? How do you know?

6. `shellSort2`

Write a method named `shellSort2` that performs the shell sort algorithm on an array of integers. Your code should modify the algorithm shown in class by choosing a different set of "gaps" to use. Create an internal array of descending gap values and sort the array by each gap value until it is fully sorted.

For example, if the array below is used with gaps of `{5, 3, 1}`, then the algorithm will perform the following manipulations of the array:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
Value	27	63	17	72	33	58	14	9	26	61	41	8	1	22	3	12
Gap 5	12	8	1	22	3	27	14	9	26	33	41	63	17	72	61	58
Gap 3	12	3	1	14	8	26	17	9	27	22	41	61	33	72	63	58
Gap 1	1	3	8	9	12	14	17	22	26	27	33	41	58	61	63	72

- Do you notice a difference in runtime when you tweak the gaps used? What gaps work well?

7. `priorityQueueSort`

Write a method named `priorityQueueSort` that sorts an array of integers by adding all of the integers to a `PriorityQueue` collection, then removing them all, which will cause them to come out in sorted order. As you implement the method, consider the following questions:

- How does the runtime of this sort compare to that of the other algorithms learned so far?
- What is the Big-Oh of this sorting technique, in terms of runtime?
- How much memory is needed for this algorithm, beyond the memory used by the array passed in?

CSE 373 Section Handout #8
Solutions

1. bubble sort

```
index        0    1    2    3    4    5    6    7
original {29, 17,  3, 94, 46,  8, -4, 12}
sweep 1  {17,  3, 29, 46,  8, -4, 12, 94}
sweep 2  { 3, 17, 29,  8, -4, 12, 46, 94}
sweep 3  { 3, 17,  8, -4, 12, 29, 46, 94}
sweep 4  { 3,  8, -4, 12, 17, 29, 46, 94}
sweep 5  { 3, -4,  8, 12, 17, 29, 46, 94}
sweep 6  {-4,  3,  8, 12, 17, 29, 46, 94}
```

2. selection sort

```
index        0    1    2    3    4    5    6    7    8
original {14, 25, 95,  0, 17, -2, 13, 56, 34}
pass 1   {-2, 25, 95,  0, 17, 14, 13, 56, 34}
pass 2   {-2,  0, 95, 25, 17, 14, 13, 56, 34}
pass 3   {-2,  0, 13, 25, 17, 14, 95, 56, 34}
pass 4   {-2,  0, 13, 14, 17, 25, 95, 56, 34}
pass 5   {-2,  0, 13, 14, 17, 25, 95, 56, 34}
pass 6   {-2,  0, 13, 14, 17, 25, 95, 56, 34}
pass 7   {-2,  0, 13, 14, 17, 25, 34, 56, 95}
pass 8   {-2,  0, 13, 14, 17, 25, 34, 56, 95}
```

3. insertion sort

```
index        0    1    2    3    4    5    6    7    8    9
original {  6,   7,   4, 11,   8,   1, 10,   3,   5,   2}
pass 1   {  6,   7,   4, 11,   8,   1, 10,   3,   5,   2}
pass 2   {  4,   6,   7, 11,   8,   1, 10,   3,   5,   2}
pass 3   {  4,   6,   7, 11,   8,   1, 10,   3,   5,   2}
pass 4   {  4,   6,   7,   8, 11,   1, 10,   3,   5,   2}
pass 5   {  1,   4,   6,   7,   8, 11, 10,   3,   5,   2}
pass 6   {  1,   4,   6,   7,   8, 10, 11,   3,   5,   2}
pass 7   {  1,   3,   4,   6,   7,   8, 10, 11,   5,   2}
pass 8   {  1,   3,   4,   5,   6,   7,   8, 10, 11,   2}
pass 9   {  1,   2,   3,   4,   5,   6,   7,   8, 10, 11}
```

4. shell sort

```
index        0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16
original {  7,   1,   6, 12, 18,   8,   4, 21,   2, 30, -1,   9, -3, 10, 11, 27, 14}
gap=8    {  2,   1, -1,   9, -3,   8,   4, 21,   7, 30,   6, 12, 18, 10, 11, 27, 14}
gap=4    {-3,   1, -1,   9,   2,   8,   4, 12,   7, 10,   6, 21, 14, 30, 11, 27, 18}
gap=2    {-3,   1, -1,   8,   2,   9,   4, 10,   6, 12,   7, 21, 11, 27, 14, 30, 18}
gap=1    {-3, -1,   1,   2,   4,   6,   7,   8,   9, 10, 11, 12, 14, 18, 21, 27, 30}
```

CSE 373 Section Handout #8
Solutions, continued

5.
```
public static void dualSelectionSort(int[] a) {
    for (int i = 0; i < a.length / 2; i++) {
        int min = i;
        int max = a.length - 1 - i;
        for (int j = i; j < a.length - i; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
            if (a[j] > a[max]) {
                max = j;
            }
        }

        if (max == i) {    // fixes the tricky max=i case
            max = min;
        }

        swap(a, i, min);
        swap(a, a.length - 1 - i, max);
    }
}
```

6.
```
public static void shellSort2(int[] a) {
    int[] gaps = {5, 3, 1};
    for (int gap : gaps) {
        for (int i = gap; i < a.length; 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;
        }
    }
}
```

7.
```
public static void priorityQueueSort(int[] a) {
    Queue<Integer> pq = new PriorityQueue<Integer>(a.length + 10);
    for (int n : a) {
        pq.add(n);
    }
    for (int i = 0; i < a.length; i++) {
        a[i] = pq.remove();
    }
}
```