Lesson 7
In Place Sorting: *Applying Powerful Technology in a Simple Context.*

Wholeness of the Lesson

BubbleSort, SelectionSort, InsertionSort are, historically, among the first sorting algorithms to be discovered. They are easy to understand. As a result, it is easy to apply the tools of asymptotic analysis to understand and compare their level of performance. *Science of Consciousness:* Likewise, to comprehend and get the full value of our own consciousness, we take some time to experience consciousness in its simplest state, free from the complexities of thoughts and perceptions. Contact with this simplest state nourishes all levels of the functioning of awareness, bringing more success and fulfillment to the field of action.

1

Analysis of Simple Sorting Algorithms

- SelectionSort and InsertionSort are among the simplest sorting methods and have straight forward analysis of running time. For each, we will consider best case, worst case, and average case running times.

2

Selection Sort Example

9	4	12	7	2
2	4	12	7	9
2	4	12	7	9
2	4	7	12	9
2	4	7	9	12

3

SelectionSort

```
Algorithm SelectionSort (arr)
 Input Array arr
 Output elements in arr are in sorted order

   for i ← 0 to arr.length - 2  do
      min ← arr[i]
      minIndex ← i
      for j ← i + 1 to arr.length - 1 do // find next smallest element
         if arr[j] < min then
            min ← arr[j]
            minIndex ← j
      if i != minIndex then
         swapElements(arr, i, minIndex)

Algorithm swapElements(arr, i,  j )
   temp ← arr[i]
   arr[i] ← arr[j]
   arr[j] ← temp
```


4

Analysis of SelectionSort

- If n is the number of items in the array, there are n-1 comparisons on the first pass, n-2 on the second, and so on. The formula for the sum of such a series is: $(n–1) + (n–2) + (n–3) + ... + 1 = n*(n–1)/2$
  Thus, the algorithm makes $\Theta(n^2)$ comparisons.
- With n items, SelectionSort performs no more than n swaps. For large values of n, the comparison times will dominate, so we have to say that the SelectionSort runs in $\Theta(n^2)$ time.

5

"Every-Case" Analysis

- Because there are two loops, nested, depending on n, it is $\Theta(n^2)$.
- Running time is the same for the best and worst cases.

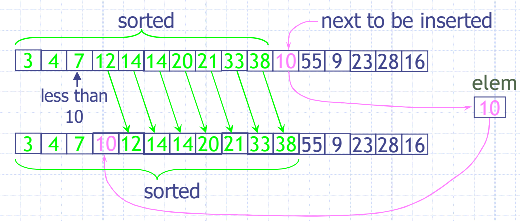
6

Insertion Sort Example

9	4	12	7	2
4	9	12	7	2
4	9	12	7	2
4	7	9	12	2
2	4	7	9	12

7

Inner While-Loop of InsertionSort



- This one step, the inner while-loop, could make O(i) shifts in the worst case

8

InsertionSort

Algorithm *InsertionSort*(*arr*)
Input Array *arr*
Output elements in *arr* are in sorted order

for $i \leftarrow 1$ **to** *arr.length* - 1 **do**
  $j \leftarrow i$
  insertElem ← *arr*[*i*]
  while $0 < j \wedge insertElem < arr[j-1]$ **do**
    arr[*j*] ← *arr*[*j* - 1]   *// shift element to right*
    $j \leftarrow j - 1$
  arr[*j*] ← *insertElem*

9

Analysis of InsertionSort

- How many comparisons and copies does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of n-1 comparisons on the last pass. This is 1 + 2 + 3 + … + n-1 = n*(n-1)/2. However, because on each pass an average of only half of the maximum number of items are actually compared before the insertion point is found, we can divide by 2, which gives $n^2/4$ on average.
- The number of shifts is approximately the same as the number of comparisons. However, a shift/move operation isn't as expensive as a swap. In any case, like selection sort, the insertion sort runs in $\Theta(n^2)$ time for random data.

10

Analysis of InsertionSort

- **Best-Case Analysis.** The best case for InsertionSort occurs when the input array is already sorted. In this case, the condition in the inner while loop always fails, so the code inside the loop never executes. The result is that execution time inside each outer loop is constant, and so running time is O(n).

11

Analysis of InsertionSort

- **Worst-Case Analysis.** Since there are two loops, nested, even in the worst case, the running time is only $\Theta(n^2)$. The worst case for InsertionSort occurs when the input array is reverse-sorted. In this case, in pass #i of the outer for loop the inner while loop must execute all its statements i times approximately, and so execution time is proportional to $1+2+\ldots+n-1=\Theta(n^2)$. Therefore, worst-case running time is $\Theta(n^2)$.

12

Analysis of InsertionSort

- **Average-Case Analysis.** It is reasonable to expect that typically, the inner while loop will not work as hard as it does in the worst-case. As mentioned earlier, on average there are $n^2/4$ comparisons. So on average, InsertionSort runs in $\Theta(n^2)$.

13

Comparing Performance of Simple Sorting Algorithms

- Swaps are more expensive than shifts/moves. Notice that swaps involve roughly eight primitive operations. This is more costly than shifting (which takes about five).
- Also, insertion sort does, on average, half as many key comparisons. Demos give empirical data for comparison.
- BubbleSort performs (on average) $\Theta(n^2)$ swaps whereas SelectionSort performs only $O(n)$ swaps, and InsertionSort does not perform any swaps at all (it shifts right which takes about half as much time as a swap). This difference explains why BubbleSort is so much slower than the other two. (Empirical studies show BubbleSort is 5 times slower than InsertionSort and 40% slower than SelectionSort and that InsertionSort is 3.5 times faster that SelectionSort on average.)

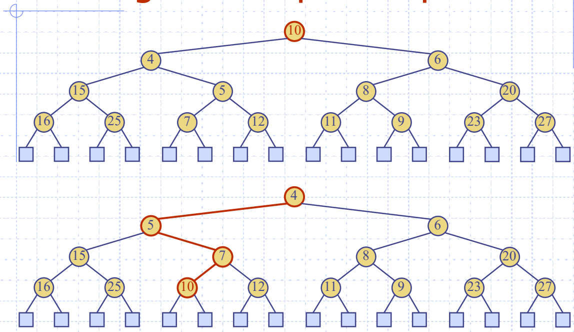
14

Main Point

1. Using the tools of asymptotic analysis, we find that, in the worst case, SelectionSort and InsertionSort run in $O(n^2)$ time, so performance of both algorithms is about the same (i.e., asymptotically equivalent). A finer analysis, which computes the number of comparisons and swaps (SelectionSort) versus comparisons and shifts (InsertionSort) performed by each algorithm, provides an account for why InsertionSort is 3.5 times faster than SelectionSort. *Science of Consciousness*: This analysis illustrates the principle that deeper levels of intelligence enable one to have greater insight and greater mastery over the more expressed values of life.

15

Building the Heap Example



16

HeapSort

```
Algorithm heapsort(arr)
 Input Array arr
 Output elements in arr are in sorted order

    buildHeap (arr)  // build the heap from bottom up O(n)

    end ← arr.length-1                              O(1)
    while end > 0 do                                O(n)
        swapElements(arr, 0, end) // move max to end of heap O(n)
        end ← end - 1  // decrease size of the heap O(n)
        downHeap(arr, 0, end) // restore heap-order O(n log n)
```


17

Build the Heap from bottom

```
Algorithm buildHeap(arr)
  Input Array arr
  Output arr is a heap built from the bottom up in O(n) time
              with the root at index 0 (instead of 1)
      last ← arr.length-1;
      next ← last;
      while (next > 0) do
            downHeap( arr, parent(next), last);
            next ← next - 2;

Algorithm parent(i)
      return floor((i – 1) / 2)
```


18

Iterative Version of downHeap

Algorithm *downHeap(H, i, last)*
Input Array *H* containing a heap and *last* (index of last element of *H*)
Output H with the heap order property restored

```
property ← false
while ! property do
    maxIndex ← indexOfMax(H, i, last)  // returns i or one of children
    if maxIndex ≠ i then
        swapElements(H, maxIndex, i)  // swaps larger to parent i
        i ← maxIndex     // move down the tree/heap to max child
    else
        property ← true
```

Algorithm *swapElements(H, j, k)*

```
temp ← H[j]
H[j] ← H[k]
H[k] ← temp
```

Heaps 19

19

Helper for downHeap Algorithm

Algorithm *indexOfMax(A, r, last)*
Input array *A*, an index *r* (referencing an element of A), and *last*, the index of the last element of the heap stored in *A*
Output index of element in *A* containing the largest of *r* or *r*'s children

```
largest ← r
left ← 2*r +1
right ← left + 1
if left ≤ last ∧ A[left] > A[largest] then
    largest ← left
if right ≤ last ∧ A[right] > A[largest] then
    largest ← right
return largest
```

Heaps 20

20

BubbleSort

Algorithm *BubbleSort(arr)*

```
n ← arr.length
for i ← 0 to n - 1 do
    for j ← 0 to n – i - 2 do
        if arr[j] > arr[j + 1] then
            swapElements(arr, j, j + 1)
```

Algorithm *swapElements(arr, i, j)*

```
temp ← arr[i]
arr[i] ← arr[j]
arr[j] ← temp
```

Simple Sorting 21

21

Analysis of BubbleSort

- In general, if n is the number of items in the array, the algorithm makes n-1 comparisons on each ith run. So there are total n*(n-1) = $\Theta(n^2)$ comparisons.
- There are fewer swaps than there are comparisons because two elements are swapped only if they need to be. If the data is random, a swap is necessary about half the time on average, so there will be about $\Theta(n^2)$ swaps.

22

22

"Every Case" Analysis

- Best case for BubbleSort occurs when input is already sorted. In that case, no swaps are performed. Still, because there are nested loops, both depending on n, running time even in this case is $\Theta(n^2)$.
- The worst case for BubbleSort occurs when input array is in reverse sorted order. In that case, the maximum number of swaps are performed. But even in the worst case, since each swap requires only constant time, the analysis is the same -- running time is $\Theta(n^2)$.
- For BubbleSort, asymptotic running time in the best, average and worst case are the same. As expected, empirical tests have shown, that BubbleSort exhibits slightly faster times on sorted or nearly sorted inputs, and slower times on inputs in reverse order.

23

23

Possible Improvements

- It is possible to implement BubbleSort slightly differently so that in the best case (which means here that the input is already sorted), the algorithm runs in $\Theta(n)$ time. (Exercise)
- As the algorithm shows, at the end of iteration i, the values in arr[n-i-1] through arr[n-1] are in final sorted order. This observation can be used to shorten the inner loop. The result is to cut the running time in half (though it must still be $\Theta(n^2)$). (Exercise)

Simple Sorting 24

24

Shell Sort:

Transcending the Limitations of Incremental Sorting

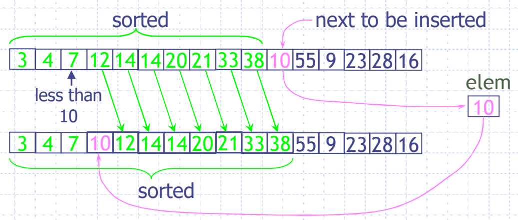
25

ShellSort

- Formulated by Donald Shell, who named the sorting algorithm after himself in 1959.
- A sorting algorithm based on InsertionSort.
- Much faster than the $O(n^2)$ sorts like SelectionSort and InsertionSort.
- Good for medium-sized arrays (as is InsertionSort generally), perhaps up to a few thousand items

26

Problem with InsertionSort: Too Many Shifts



- This one step possibly makes more shifts than necessary.

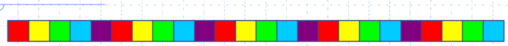
27

Problem of InsertionSort: Too Many Shifts

- Suppose a small item is on the far right. To move this small item to its proper place on the left, all the intervening items (between the place where it is and where it should be) must be shifted one space to the right. If there are i intervening items, this step takes close to i copies to handle one item. If i is close to n, then it takes O(n) to put just one item to the proper place.
- This performance could be improved if we could somehow move a smaller item many spaces to the left without shifting all the intermediate items individually.

28

ShellSort – General Description



- Essentially a segmented InsertionSort
  - Divides an array into several smaller noncontiguous segments
  - The distance between successive elements in one segment is called a *gap/Increment* (usually represented by h).
  - Each segment is sorted within itself using InsertionSort.
  - Then re-segment into larger segments (smaller gaps) and repeat sort.
  - Continue until only one segment (h= 1).

29

Sorting nonconsecutive subarrays

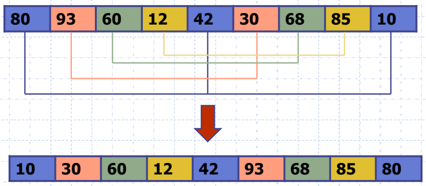
Here is an array to be sorted (numbers aren't important)

- Consider just the red locations.
- We try doing insertion sort on *just the red numbers,* as if they were the only ones in the array.
- Next do the same for just the yellow locations -- we do an insertion sort on just these numbers.
- Now do the same for each additional group of numbers.
- The resultant array is sorted *within groups,* but not overall.

30

Example

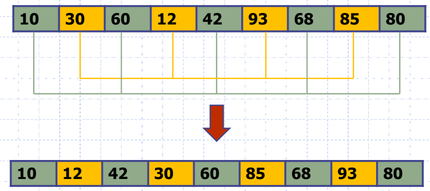
Initial Segmenting Gap = 4



31

Example

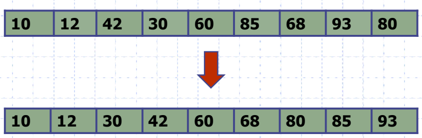
Initial Segmenting Gap = 2



32

Example

Initial Segmenting Gap = 1



33

Diminishing Gaps and N-Sorting

- In the example, we have seen an initial interval/gap of 4 cells for sorting a 9-cell array. This means that all items spaced four cells apart are sorted among themselves. We call it 4-sort. In general, applying InsertionSort on gap of N cells is called N-Sorting.
- The interval is then repeatedly reduced until it becomes 1.
- The set of intervals used in the example, (4, 2, 1) is called the *interval sequence* or *gap sequence*.
- Using this gap sequence, we can also say we did 4-sort, 2-sort, and 1-sort on the example.

34

Obtaining a Gap Sequence

- Any decreasing gap sequence will work (if the last gap is 1), but the running time depends crucially on the choice of the gap sequence.
- When the gap sequence consists of powers of 2, such as (8, 4, 2, 1) (this was Shell's original method) it can be shown that the worst-case running time is no better than InsertionSort: $O(n^2)$. Running time is improved when terms of the gap sequence are relatively prime (no common factors other than 1).

35

Added Gap Sequence

- Donald Knuth, in his discussion of Shell's Sort, recommended another sequence of gaps.

  $h_0 = 1$, $h_{j+1} = h_j * 3 + 1$

  - Find the largest h_j <= n, then start with h_j

h	3*h + 1	(h–1) / 3
1	4	
4	13	1
13	40	4
40	121	13
121	364	40
364	1093	121
1093	3280	364

For example, sorting a 1,000-element array, first needs to find a largest h <= n, which will be 364 in this case. Then, reduce the interval using the inverse of the formula given:

$h = (h-1) / 3$

This inverse formula generates the reverse sequence 364, 121, 40, 13, 4, 1.

36

ShellSort Algorithm

```
Algorithm shellSort(arr)
  Input Array arr
  Output elements of arr are rearranged into sorted order
    maxGap ← floor((arr.length-1)/3)
    h ← 1
    while (h ≤ maxGap) do  // compute the maximum gap size
        h ← h * 3 + 1  // 1,4,13,40,121,...
    while (0 < h) do
        segmentInsertionSort(arr, h)
        h ← (h - 1) / 3  // ...,121,40,13,4,1

Algorithm segmentInsertionSort(arr, gap)
   for i ← gap to arr.length - 1 do
      j ← i
      insertElem ← arr[i]
      while j > gap-1 ∧ insertElem < arr[j - gap] do
            arr[j] ← arr[j - gap]       // shift element to right
            j ← j – gap
      arr[j] ← insertElem
```


37

Running time

- Real running time of Shellsort? Although running times for ShellSort using certain gap sequences are known, finding the gap sequence that produces the best possible running time is still being researched.
- For any version of ShellSort, we do know that its average running time is $O(n^r)$ with $1 < r < 2$
- Generally speaking, ShellSort's running time is better than $O(n^2)$ but worse than $O(n \log n)$.

38

Ideal Gap Sequence

Although mathematical techniques have been developed to optimize the gap sequence used, the best gap sequences have been found just by empirical tests. Here are a few of the best known results [see https://en.wikipedia.org/wiki/Shellsort]

Concrete gaps	Worst-case time complexity	Author and year of publication
$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta(N^2)$ [when $N=2^p$]	Shell, 1959[3]
$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta(N^{3/2})$	Frank & Lazarus, 1960[7]
$1, 3, 7, 15, 31, 63, \ldots$	$\Theta(N^{3/2})$	Hibbard, 1963[8]
$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta(N^{3/2})$	Papernov & Stasevich, 1965[9]
$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta(N \log^2 N)$	Pratt, 1971[10]
$1, 4, 13, 40, 121, \ldots$	$\Theta(N^{3/2})$	Pratt, 1971[10]

39

Main Point

2. The first algorithm to overcome the limitations of the simple sorting algorithms – BubbleSort, SelectionSort, Insertion Sort – was ShellSort. The strategy used in ShellSort was to remove one of the known limitations of the InsertionSort algorithm. The technique results in a significant jump in performance. *Science of Consciousness*: This step in the history of sorting algorithms illustrates the general principle that removal of blockages to optimal functioning of a system can greatly improve its performance. This is the strategy used by TM, which results in removal of stress and significant improvements in performance in life: intelligence, efficiency, satisfaction.

40

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
selection-sort	$O(n^2)$	Okay but slow even on small inputs (<1K)
insertion-sort	$O(n^2)$ or $O(n+k)$	excellent for small inputs, fastest for 'almost' sorted
heap-sort	$O(n \log n)$	in-place, fast, excellent for large inputs (1K to M)
Shell-sort	$O(n^{3/2})$	in-place, fast, excellent for large inputs (1K to M)

41

Efficient InsertionSort on List

```
Algorithm InsertionSort(Lst)
 Input List Lst
 Output elements in Lst are in sorted order
   if Lst.size() < 2 then return
   iter ← Lst.positions()
   next ← iter.nextObject()
   while iter.hasNext() do
      prev ← next
      next ← iter.nextObject()
      insertElem ← next.element()
      curr ← next
      while ! Lst.isFirst(curr) ∧ insertElem < prev.element() do
          Lst.replaceElement(next, prev.element())  // shift element to right
          curr ← prev
          if ! Lst.isFirst(prev) then prev ← Lst.before(prev)

      Lst.replaceElement(next, insertElem)
```


42

InsertionSort

Algorithm *InsertionSort*(*arr*)
Input Array *arr*
Output elements in *arr* are in sorted order

```
for i ← 1 to arr.length - 1 do
    j ← i
    insertElem ← arr[i]
    while 0 < j ∧ insertElem < arr[j - 1] do
        arr[j] ← arr[j - 1]      // shift element to right
        j ← j – 1
    arr[j] ← insertElem
```


43

Connecting the Parts of Knowledge
With the Wholeness of Knowledge
Simple Sorting

1. Insertion Sort sorts by examining each successive value x in the input list and searches the already sorted section of the array for the proper location for x.
2. ShellSort is also a sorting algorithm which does the same steps as InsertionSort, but, before carrying out those steps, performs a number of pre-processing steps, called *n-sorting.* The result of this refinement is that, even in the worst case, good versions of ShellSort run in $O(n^r)$ for $r < 2$.
3. *Transcendental Consciousness* is the silent field of pure intelligence, the home of all knowledge, the basis of all activity.
4. *Impulses within the Transcendental field.* Contact with transcendental consciousness enlivens the support of the laws of nature for activity in life. Therefore, the "pre-processing" step necessary for success in life is *transcend.* Then activity will be smooth and successful.
5. *Wholeness moving within itself.* In Unity Consciousness, the transcendental level has already been automatically integrated with ordinary active awareness – no "pre-processing" step is needed in this state.

44