"Slow" Sorting Algorithms

Slow $O(n^2)$ algorithms:

- Bubble Sort
- Selection Sort
- Insertion Sort

Not-quite-as-slow (better than $O(n^2)$ but not quite $O(n \log n)$) algorithm:

- Shellsort

1

1

Objectives

In these lectures, we'll consider:

- "Slow" Sorting Algorithms
- "Fast" Sorting Algorithms

2

2

Bubble Sort

- Bubble Sort makes a series of sweeps across the list of data
- In each sweep, adjacent items are compared and swapped if needed
- As a result, after each sweep is completed, the largest remaining value ends up being shifted to the far right of the part of the list not yet sorted
- That largest value then joins the part of the list that is sorted
- Worst-case computational complexity is $O(n^2)$
- A Boolean can detect if no swapping is done during a sweep that allows the sort to end “early” making a best-case O(n)

3

3

Bubble Sort

```
void bubbleSort(int data[], int size) {
   int temp;
   bool has_swapped;

   for (int i = size; i > 0; i--) {
      has_swapped = false;
      for (int j = 0; j < i-1; j++)
         if (data[j] > data[j+1]) {
            swap(data[i], data[j]);
            has_swapped = true;
         }
      if (!has_swapped) { break; }
}  }
```

4

4

Selection Sort

- Like Bubble Sort, list is split into Sorted and Unsorted parts
- For each sweep, the smallest remaining value in the Unsorted part is found and marked for swapping
- At the end of the sweep, one swap is done, moving the marked value to the far left of the Unsorted area
- The far left of the Unsorted area is appended to the Sorted area, and the next sweep begins
- This means the total number of swaps is no greater than the number of items in the list

5

5

Selection Sort

```
void selectionSort(int data[], int size) {
   int temp = 0;

   for (int i = 0; i < size; i++) {
      int smallest_index = i;
      // find index of smallest unsorted element
      for (int j = i + 1; j < size; j++)
         if (data[smallest_index] > data[j])
            smallest_index = j;

      //swap smallest index with value at i (if necessary)
      if (smallest_index != i)
         swap(data[i], data[smallest_index];
   }
}
```

6

6

Insertion Sort

- Like Bubble Sort, the list will have Sorted and Unsorted parts
- For each sweep, the leftmost value in the Unsorted area is compared to its left neighbor in the Sorted area and swapped as needed
- The unsorted element will be compared and swapped with its left neighbor until the left neighbor is no longer greater than the unsorted element
- That element is then in the correct position in the Sorted area, and the next leftmost Unsorted element is processed

7

7

Insertion Sort

```
void insertionSort(int data[], int size) {
   int temp;

   for (int i = 0; i < size - 1; i++) {
      for (int j = i + 1; j > 0; j--) {
         if (data[j] < data[j-1]) {
            swap(data[j], data[j-1]);
         else break;
         }
      }
   }
}
```

8

8

Shellsort

- Named after a guy named Shell ☺, who published the algorithm in 1959 when he get his PhD at U of Cincinnati
- Parts of the array are Insertion-sorted separately based on a “gap length,” the distance between elements in the array
- After each sub-array in the list is sorted, the gap length is shortened and Insertion sort is performed again
- Since the sub-arrays are sorted, the number of swaps needed to sort the larger “partially” sorted sub-arrays based on smaller gap length is less than it would be for random values
- The gap length eventually becomes 1 (no gap) and the array is sorted

9

9

Shellsort

```
void shellSort(int data[], int size) {
    int temp;
    // Start with a large gap, then reduce the gap
    for (int gap=size/2; gap>0; gap /= 2) {
        // Do a gapped Insertion sort for this gap size
        for (int i=gap; i<size; i++) {
            temp = data[i];
            for (int j=i; j>=gap && data[j-gap] >temp; j -= gap)
                data[j] = data[j-gap];
            data[j] = temp;
        }
    }
}
```

10

10