CSE 373 Lecture 15: Sorting

- Today's Topics:
  - Elementary Sorting Algorithms:
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
  - Shellsort
- Covered in Chapter 7 of the textbook

Sorting: Definitions

- Input: You are given an array A of data records, each with a key (which could be an integer, character, string, etc.).
  - There is an *ordering* on the set of possible keys
  - You can compare any two keys using <, >, ==
- For simplicity, we will assume that A[i] contains only one element – the key
- Sorting Problem: Given an array A, output A such that:
  For any i and j, if $i < j$ then $A[i] \leq A[j]$
- *Internal sorting* → all data in memory, *External* → data on disk

Why Sort?

- Sorting algorithms are among the most frequently used algorithms in computer science
  - Crucial for efficient retrieval and processing of large volumes of data E.g. Database systems
- Allows binary search of an N-element array in O(log N) time
- Allows O(1) time access to *k*th largest element in the array for any *k*
- Allows easy detection of any duplicates

Sorting: Things to Think about…

- Space: Does the sorting algorithm require extra memory to sort the collection of items?
  - Do you need to copy and temporarily store some subset of the keys/data records?
  - An algorithm which requires O(1) extra space is known as an **in place** sorting algorithm
- Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
  - E.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
  - Extremely important property for databases
  - A **stable sorting algorithm** is one which does not rearrange the order of duplicate keys

Sorting 101: Bubble Sort

- Idea: "Bubble" larger elements to end of array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
  - Repeat from first to end of unsorted part
- Example: Sort the following input sequence:
  - 21, 33, 7, 25

Sorting 101: Bubblesort

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }
void bubble( int A[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=1;j<(n-i);j++) {
      /* If adjacent items out of order, swap */
      if( A[j-1] > A[j] ) SWAP(A[j-1],A[j]); }
  }
}
```

- Stable? In place? Running time = ?

Sorting 102: Selection Sort

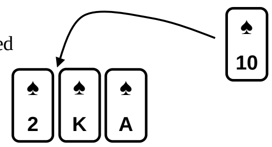
- Bubblesort is stable and in place, but $O(N^2)$ – can we do better by moving items more than 1 slot per step?
- Idea: Scan array and select smallest key, swap with A[1]; scan remaining keys, select smallest and swap with A[2]; repeat until last element is reached.
- Example: Sort the following input sequence:
  - 21, 33, 7, 25
- Is selection sort stable (suppose you had another 33 instead of 7)? In place?
- Running time = ?

Sorting 102: Selection Sort

- Bubblesort is $O(N^2)$ – can we do better by moving items more than 1 slot per step?
- Idea: Scan array and select smallest key, swap with A[1]; scan remaining keys, select smallest and swap with A[2]; repeat until last element is reached.
- Example: Sort the following input sequence:
  - 21, 33, 7, 25
- NOT STABLE. In place (extra space = 1 temp variable).
- Running time = N steps with N-1, …, 1 comparisons = N-1 + … + 1 = $O(N^2)$

Sorting 103: Insertion Sort

- What if first *k* elements of array are already sorted?
  - E.g. <u>4, 7, 12,</u> 5, 19, 16
- Idea: Can *insert* next element into proper position and get k+1 sorted elements, *insert* next and get k+2 sorted etc.
  - <u>4, 5, 7, 12,</u> 19, 16
  - <u>4, 5, 7, 12, 19,</u> 16
  - <u>4, 5, 7, 12, 16, 19</u> Done!
  - Overall, N-1 passes needed
  - Similar to card sorting…
  - Start with empty hand
  - Keep inserting…



Sorting 103: Insertion Sort

```
void InsertionSort( ElementType A[ ], int N ) {
  int j, P; ElementType Tmp;
  for( P = 1; P < N; P++ ) {
      Tmp = A[ P ];
      for( j = P; j > 0 && A[ j - 1 ] > Tmp;j-- )
              A[ j ] = A[ j - 1 ];
      A[ j ] = Tmp;
  }
}
```

- Is Insertion sort in place? Stable?
- Running time = ?

Sorting 103: Insertion Sort

```
void InsertionSort( ElementType A[ ], int N ) {
  int j, P; ElementType Tmp;
  for( P = 1; P < N; P++ ) {
      Tmp = A[ P ];
      for( j = P; j > 0 && A[ j - 1 ] > Tmp;j-- )
              A[ j ] = A[ j - 1 ];
      A[ j ] = Tmp;
  }
}
```

- Insertion sort → in place (O(1) space for Tmp) and stable
- Running time: Worst case → reverse order input = $\Theta(N^2)$
  - Best case → input already sorted = O(N).

Lower Bound on Simple Sorting Algorithms

- An *inversion* is a pair of elements in wrong order
  - i < j but A[i] > A[j]
- Our simple sorting algorithms so far swap adjacent elements (explicitly or implicitly) → removes 1 inversion
  - Their running time is proportional to number of inversions in array
- Given N distinct keys, total of N(N-1)/2 possible inversions. Average list will contain half this number of inversions = N(N-1)/4
  - Average running time of Insertion sort is $\Theta(N^2)$
- Any sorting algorithm that swaps adjacent elements requires $\Omega(N^2)$ time → each swap removes only one inversion

Shellsort: Breaking the Quadratic Barrier

- Named after Donald Shell – first algorithm to achieve $o(N^2)$
  - Running time is $O(N^x)$ where x = 3/2, 5/4, 4/3, …, or 2 depending on "increment sequence"
- Idea: Use an *increment sequence* $h_1 < h_2 < \ldots < h_t$
  - Start with k = t
  - Sort all subsequences of elements that are h_k apart so that $A[i] \le A[i+h_k]$ for all i → known as an h_k*-sort*
  - Go to next smaller increment h_{k-1} and repeat until k = 1
- Example: Shell's original sequence: $h_t = N/2$ and $h_k = h_{k+1}/2$
  - Sort 21, 33, 7, 25
  - Try it! (What is the increment sequence?)

Shellsort: Breaking the Quadratic Barrier

- Named after Donald Shell – first algorithm to achieve $o(N^2)$
  - Running time is $O(N^x)$ where x = 3/2, 5/4, 4/3, …, or 2 depending on "increment sequence"
- Idea: Use an *increment sequence* $h_1 < h_2 < \ldots < h_t$
  - Start with k = t
  - Sort all subsequences of elements that are h_k apart so that $A[i] \le A[i+h_k]$ for all i → known as an h_k*-sort*
  - Go to next smaller increment h_{k-1} and repeat until k = 1
- Example: Shell's original sequence: $h_t = N/2$ and $h_k = h_{k+1}/2$
  - Sort 21, *33*, 7, *25* (N = 4, increment sequence = 2, 1)
  - 7, *25*, 21, *33* (after 2-sort)
  - 7, 21, 25, 33 (after 1-sort)

Shellsort

```
void Shellsort( ElementType A[ ], int N ){
  int i, j, Increment; ElementType Tmp;
  for( Increment = N/2; Increment > 0; Increment /= 2 )
    for( i = Increment; i < N; i++ )                {
      Tmp = A[ i ];
      for( j = i; j >= Increment; j -= Increment )
            if( Tmp < A[ j - Increment ] )
                  A[ j ] = A[ j - Increment ];
            else
            break;
      A[ j ] = Tmp;
    }
}
```

- Running time = ? (What is the worst case?)

Answer and further analysis in next class…

Also in the next class, the crème de la crème:

Heapsort, Mergesort, and Quicksort

To Do:

If you can't wait, read chapter 7

If you can, read chapter 7 anyway…