Data Structures and Algorithms 2

Prof. Ahmed Guessoum

The National Higher School of AI

Chapter 7

Sorting

Preliminaries

- We discuss the problem of sorting an array of elements.
- We will assume that the array contains only integers, although the code will allow more general objects.
- We will assume that the entire sort can be done in main memory ➔ the number of elements is relatively small (less than a few million)
- External sorting (which must be done on disk or tape), will be discussed at the end of the chapter.
- We will assume the existence of the "<" and ">" operators (besides the assignment operator). They can be used to place a consistent ordering on the input. Sorting under these conditions is known as **comparison-based sorting**.

Sorting in the STL

- The interface that will be used is not the same as in the STL sorting algorithms.
- In STL, sorting (generally **quicksort**) is accomplished by use of the function template sort.

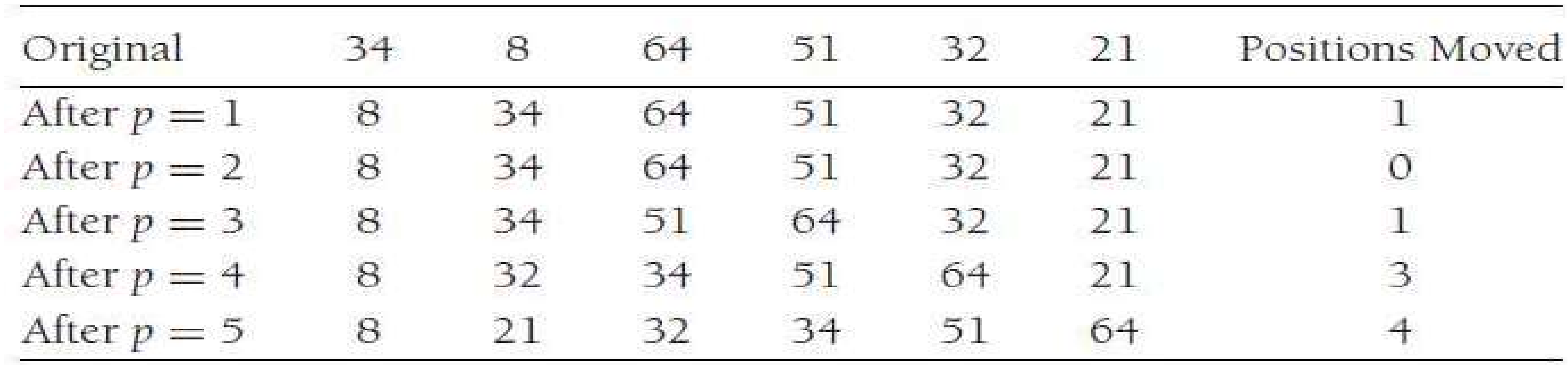
```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator cmp );
```

- The iterators must support random access.
- The sort algorithm does not guarantee that equal items retain their original order (if that is important, use stable_sort instead of sort).

```
std::sort( v.begin( ), v.end( ) );
                    // sort the entire container, v, in nondecreasing order
std::sort( v.begin( ), v.end( ), greater<int>{ } );
                    // sort container v in nonincreasing order
std::sort( v.begin( ), v.begin( ) + ( v.end( ) - v.begin( ) ) / 2 );
                    // sort first half of container v in nondecreasing order
```

Insertion Sort

- It is one of the simplest sorting algorithms.
- *Insertion sort* consists of $N-1$ **passes.**
- For pass $p = 1$ through $N-1$, *insertion sort* ensures that the elements in positions 0 through p are in sorted order
  - It makes use of the fact that elements in positions 0 through $p-1$ are already known to be in sorted order.
  - In pass p , we move the element in position p left until its correct place is found among the first $p+1$ elements.
  - The element in position p is moved to *tmp*, and all larger elements (prior to position p) are moved one spot to the right. Then *tmp* is moved to the correct spot at the end.



Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

```
//  Simple insertion sort.
template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
        for( int p = 1; p < a.size( ); ++p ) {
            Comparable tmp = std::move( a[ p ] );
            int j;
            for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
                  a[ j ] = std::move( a[ j - 1 ] );
            a[ j ] = std::move( tmp );
      }
}
```

Analysis of Insertion Sort

- Because of the nested loops, each of which can take N iterations, insertion sort is $O(N^2)$.
- A precise calculation shows that the number of tests in the inner loop is at most $p + 1$ for each value of p. Summing over all p gives a total of

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

- If the input is pre-sorted (or almost sorted), the running time is $O(N)$, because the test in the inner for loop always fails immediately

- An **inversion** in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$.
- In the previous example: the input list 34, 8, 64, 51, 32, 21 had nine inversions, namely (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), and (32, 21).
- This is exactly the number of swaps that needed to be (implicitly) performed by insertion sort.
- This is always the case, because swapping two adjacent elements that are out of place removes exactly one inversion, and a sorted array has no inversions.
- Since there is $O(N)$ other work involved in the algorithm, the running time of insertion sort is $O(I + N)$, where I is the number of inversions in the original array
- Thus, insertion sort runs in linear time if the number of inversions is $O(N)$.

- We can compute precise bounds on the average running time of insertion sort by computing the average number of inversions in a permutation.
- We will assume that there are no duplicate elements
- Using this assumption, we can assume that the input is some permutation of the first N integers
- Under these assumptions, we have the following theorem:

Theorem 7.1: The average number of inversions in an array of N distinct elements is $N(N-1)/4$. (Proof: See textbook.)

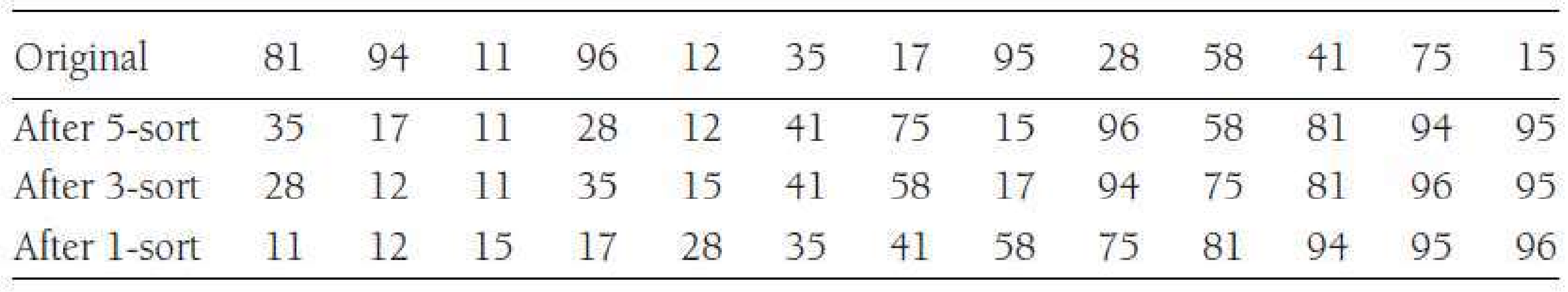
- This theorem implies that insertion sort is quadratic on average. It also provides a very strong lower bound about any algorithm that only exchanges adjacent elements.

Theorem 7.2: Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average. (Proof: See textbook.)

Shellsort

- Shellsort (Donald Shell) was one of the first algorithms to break the quadratic time barrier
- It works by comparing elements that are distant; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared.
- For this reason, Shellsort is sometimes referred to as **diminishing increment sort**.
- Shellsort uses a sequence, $h_1, h_2, \ldots, h_t$, called the **increment sequence**.
- Any increment sequence will do as long as $h_1 = 1$, but some choices are better than others.
- After a *phase,* using some increment h_k, for every i, we have $a[i] \leq a[i + h_k]$
- <u>Important property of Shellsort</u>: if you h_{k-1}-sort an h_k-sorted file, then it remains h_k-sorted

- The general strategy to h_k-sort is for each position, *i*, sort the subarray starting at i of elements obtained by increments h_k; then take the next (smaller) h_k and so on.
- Suppose the increment sequence is 5, 3, 1



Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

- A popular (but poor) choice for increment sequence is to use the sequence suggested by Shell: h_t = floor(N / 2), and h_k = floor(h_{k+1} / 2).

```
// Shellsort routine using Shell's increments (better increments are possible)
template <typename Comparable>
void shellsort( vector<Comparable> & a )
{
        for( int gap = a.size( ) / 2; gap > 0; gap /= 2 )
                for( int i = gap; i < a.size( ); ++i )
                {
                        Comparable tmp = std::move( a[ i ] );
                        int j = i;
                        for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )
                                a[ j ] = std::move( a[ j - gap ] );
                        a[ j ] = std::move( tmp );
                }
}
```

Worst-Case Analysis of Shellsort

- The running time of Shellsort depends on the choice of increment sequence, and the proofs can be rather involved.
- The average-case analysis of Shellsort is a long-standing open problem, except for the most trivial increment sequences.

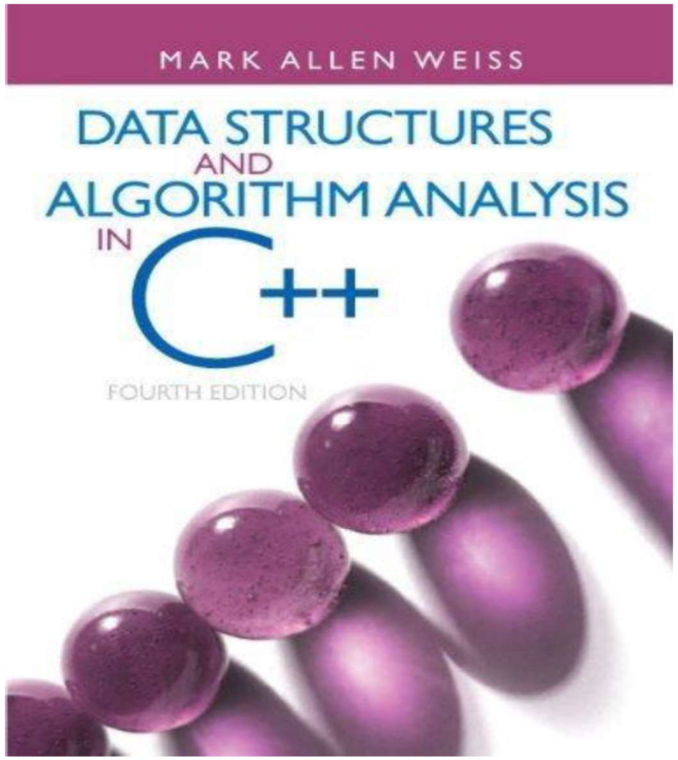
Theorem 7.3: The worst-case running time of Shellsort using Shell’s increments is $\Theta(N^2)$.

- Hibbard suggested a slightly different increment sequence, which gives better results in practice (and theoretically): 1, 3, 7, . . . , $2^k - 1$.
- Key difference: consecutive increments have no common factors.
- For this increment sequence, we have the following theorem:

Theorem 7.4: The worst-case running time of Shellsort using Hibbard’s increments is $\Theta(N^{3/2})$.

- The performance of Shellsort is quite acceptable in practice, even for N in the tens of thousands. The simplicity of the code makes it the algorithm of choice for sorting up to moderately large input.

Slides based on the textbook



Mark Allen Weiss, (2014) Data Structures and Algorithm Analysis in C++, 4th edition, Pearson.

Acknowledgement: This **course PowerPoints** make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.