

# Chapter 9

## Sorting Algorithms

### Terminology

- Internal – done totally in main memory.
- External – uses auxiliary storage (disk).
- Stable – retains original order if keys are the same.
- Oblivious – performs the same amount of work regardless of the actual input.
- Sort by address – uses indirect addressing so the record (structure) doesn't have to be moved.

### Selection Sort

- Select elements one at a time and place in proper final position.
- Repeatedly find the smallest.
- 3 6 43 1 9  
1 6 43 3 9  
1 3 43 6 9  
1 3 6 43 9  
1 3 6 9 43
- Code is shown in Figure 1.
- Analysis:  $n + n - 1 + n - 2 + \dots + 1 = n(n - 1) / 2$
- Only  $n$  moves – use when records (structure) are long.
- Requires  $n^2 / 2$  compares even when already sorted - *oblivious*.

```
template<class T>
void SelectionSort(T a[], int n)
{// Sort the n elements a[0:n-1].
  for (int size = n; size > 1; size--) {
    int j = Max(a, size);
    Swap(a[j], a[size - 1]);
  } // for
} // SelectionSort
```

Figure 1 Selection Sort

### Bubble Sort

- Compares adjacent elements – exchanges if out of order.
- Lists get smaller each time – at least one is placed in final order.
- Place of last swap is as much as you have to look at.
- Code is shown in Figure 2.
- Analysis:  
 $n + n - 1 + n - 2 + \dots + 1 = n(n - 1) / 2$

```
template<class T>
void BubbleSort(T a[], int n)
{// Sort a[0:n - 1] using bubble sort.
  for (int i = n; i > 1; i--)
    for (int j = 0; j < i - 1; j++)
      if (a[j] > a[j+1]) Swap(a[j], a[j+1]);
} // BubbleSort
```

Figure 2 Bubble Sort

## Insertion Sort

Sorts by inserting records into an already existing sorted file.

- Two groups of keys - sorted and unsorted.
- Code is shown in Figure 3.
- Insert  $n$  times - each time move  $\frac{1}{2}$  elements to insert.

- The number of elements in the list changes so

$$\frac{1}{2}(1 + 2 + 3 + \dots + n) = \frac{1}{2} \times \frac{1}{2}n(n+1) = \frac{1}{4}n^2$$

- 11 5 17 1 21
- 5 11 17 1 21
- 5 11 17 1 21
- 1 5 11 17 21
- 1 5 11 17 21

```
template<class T>
void InsertionSort(T a[], int n)
{// Sort a[0:n-1].
  for (int i = 1; i < n; i++)
  { T t = a[i];
    for (int j = i - 1; j >= 0 && t < a[j]; j--)
      a[j+1] = a[j];
    a[j+1] = t;
  } // for
} // InsertionSort
```

Figure 3 Insertion Sort

- Performs better when the degree of unsortedness is low – recommended for data that is nearly sorted.
- Improvements
  1. Use binary search  $O(n \log n)$  compares, but number of moves doesn't change so no real gain.
  2. Linked list storage – can't use binary search – still  $O(n^2)$ .
- Could use sentinel containing the key search until  $p \rightarrow \text{info.key} \geq q \rightarrow \text{info.key}$
- **Sentinel** is extra item added to one end of the list so ensure that the loop terminates without having to include a separate check.

## Shell Sort

- A subquadratic algorithm whose code is only slightly longer than insertion sort, making it the simplest of the faster algorithms.
- Avoid large amounts of data movement by:
  - Comparing elements that are far apart.
  - Comparing elements that are less far apart.
  - Gradually shrinks toward insertion sort.
- Consider Figure 4, which shows an example of shell sort with the increment sequence of {1, 3, 5}.

<b>Original</b>	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

Figure 4 Shell Sort

- Shell sort is known as a *diminishing gap sort*.
- The code for Shell sort is shown in Figure 5.
- Worst case running time is  $O(n^2)$ .
- The average running time appears to be between  $O(n^{5/4})$ – $O(n^{7/6})$  if we use a gap of 2.2.

```

template<class T>
void ShellSort (T a[], int length )
{ for ( int gap = length / 2; gap > 0;
      gap = gap == 2 ? 1 : static_cast<int>( gap / 2.2 ) )
  for ( int i = gap; i < length; i++ )
  {   T tmp = a[i];
      for ( ; j >= gap && tmp < a[j-gap]; j -= gap )
          a[j] = a[j-gap];
      a[j] = tmp;
  } // for
} // ShellSort

```

Figure 5 Shell Sort

## Merge Sort

- Chop the lists into two sublists. Sort the two pieces. Combine by merging. (Some techniques just get sublists of larger and larger powers of two.)
- The pieces may **also** be sorted via MergeSort, so it is recursive.
- What would the code look like for MergeSort?
- The code for Merge is shown in Figure 6.
- In merging sublists of length  $n$ , clearly no more than  $2n$  compares are required. Actually it is less than this, but it is easy to count this way.
- Each level takes exactly  $n$  compares and there are  $\log n$  levels, the complexity is  $O(n \log n)$ .
- A closer count is  $n \log n - 1.25n + 1$  (by running test cases)
- If linked lists, no problem with storage space. If we have an array of items to be stored, the MergeSort requires an auxiliary storage array.

```

template<class T>
void Merge(T c[], T d[], int l, int m, int r)
{// Merge c[l:m] and c[m:r] to d[l:r].
  int i = l, // cursor for first segment
      j = m+1, // cursor for second
      k = l; // cursor for result

  while ((i <= m) && (j <= r))
    if (c[i] <= c[j]) d[k++] = c[i++];
    else d[k++] = c[j++];

  if (i > m) for (int q = j; q <= r; q++)
              d[k++] = c[q];
  else for (int q = i; q <= m; q++)
          d[k++] = c[q];
} // Merge

```

Figure 6 Merge Sort

## Quick Sort

- Partition the set into two sets: those elements less than  $j$  and those elements greater than  $j$ .  $j$  is called the pivot.
- Often done as: Use two pointers – top pointer looks for a value smaller than  $j$ , bottom pointer looks for a value larger than  $j$ . Then interchange. This does only a third as many swaps.

- Apply recursively.
- The code for QuickSort is shown in Figure 7.
- Analysis: At each level, all elements of the array are examined. The number of levels depends on how equally the pieces are divided. Best case:  $\log n$  levels yielding  $O(n \log n)$ .
- Worst case: let  $n$  be the size of the array to be sorted:  
 $C(n) = n - 1 + C(n - 1) = n(n - 1) / 2$
- Space requirement: depends on recursive stacking.
- Improvements
  1. Use median of three so don't get a bad pivot.
  2. Use sentinel at each end so don't have to check (avoid left and right crossing)
  3. Switch to insertion sort when size gets small - can do one big insertion sort on all.

```

template<class T>
void QuickSort( T a[], int l, int r )
{ if ( l >= r ) return;
  int i = l,    // left-to-right cursor
      j = r + 1; // right-to-left cursor
  T pivot = a[l];
  while ( true ) {
    do { // find >= element on left side
      i = i + 1;
    } while ( a[i] < pivot );
    do { // find <= element on right side
      j = j - 1;
    } while ( a[j] > pivot );
    if ( i >= j ) break; // swap pair not found
    Swap( a[i], a[j] );
  } // while
  a[l] = a[j];
  a[j] = pivot;
  QuickSort( a, l, j-1 ); // sort left segment
  QuickSort( a, j+1, r ); // sort right segment
} // QuickSort

```

Figure 7 Quick Sort

## Final Thought

- In Figure 8, notice that QuickSort and MergeSort have a similar program structure.
- Both have  $O(n)$  work to either
  1. Divide into chunks or
  2. Put the chunks back together.
- The pictures we draw (for expected case) look the same.
- The formula analysis looks the same.
- We see it doesn't matter whether we do the work before the recursion or after. The work is the same.

```

QuickSort(a[],low,high) {
  pivot = partition(a,low,high)
  QuickSort(a,low,pivot-1)
  QuickSort(a,pivot+1,high)
} // QuickSort

MergeSort(a[],low,high) {
  mid=(low+high)/2
  MergeSort(a,low,mid)
  MergeSort(a,mid+1,high)
  Merge(a,low,mid,high)
} // MergeSort

```

Figure 8 QuickSort and MergeSort

## Quickselect

- Find the  $k^{\text{th}}$  smallest element in an array of  $N$  items.
- We can do better than sorting the array first.

- The solution is to have an algorithm that is similar to QuickSort but with only one recursive call.
- The steps of the algorithm are as follows:
  1. If the number of elements in  $S$  is 1, return the single element in  $S$ .
  2. Pick any element  $v$  in  $S$ . It is the pivot.
  3. Partition  $S - \{v\}$  into  $L$  and  $R$ , exactly as was done with quicksort.
  4. If  $k \leq$  number of elements in  $L$ , the item we are searching for must be in  $L$ . Call *Quickselect*(  $L, k$  ) recursively. Otherwise, if  $k$  is exactly equal to 1 more than the number of items in  $L$ , the pivot is the  $k^{\text{th}}$  smallest element, and we can return it as the answer. Otherwise, the  $k^{\text{th}}$  smallest element lies in  $R$ , and it is the  $(k - |L| - 1)^{\text{th}}$  smallest element in  $R$ . Again, we can make a recursive call and return the result.
- Notice that only one recursive call is made.
- If the pivot is chosen correctly, it can be shown, that even in the worst case, the running time is linear.

## Indirect Sorting

- If we are sorting an array whose elements are large, copying the items can be very expensive.
- We can get around this by doing *indirect sorting*.
  - We have an additional array of pointers where each element of the pointer array points to an element in the original array.
  - When sorting, we compare keys in the original array but we swap the element in the pointer array.
- This saves a lot of copying at the expense of indirect references to the elements of the original array.