SAMPLE OF THE STUDY MATERIAL

PART OF CHAPTER 6

Sorting Algorithms

6.0 Introduction

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the list. For typical sorting algorithms good behavior is O(nlogn) and bad behavior is $O(n^2)$.
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only O(1) memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

6.1 Bubble sort

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. First pass bubble out the largest element and places in the last position and second pass place the second largest in the second last position and so on. Thus, in the last pass smallest items is placed in the first position. Because, it only uses comparisons to operate on elements, **it is a comparison sort.**

```
for i = 1:n,
   swapped = false
   for j = n:i+1,
      if a[j] < a[j-1],
         swap a[j,j-1]
```

```
        swapped = true
    → invariant: a[1..i] in final position
    break if not swapped
end
```

Example:

Let us take the array of numbers "5 1 4 2 9", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared.

First Pass: Considering length n

(**5 1** 4 2 9) ⟶ (**1 5** 4 2 9), Here, algorithm compares the first two elements, and swaps them.
(1 **5 4** 2 9) ⟶ (1 **4 5** 2 9), Swap since 5 > 4
(1 4 **5 2** 9) ⟶ (1 4 **2 5** 9), Swap since 5 > 2
(1 4 2 **5 9**) ⟶ (1 4 2 **5 9**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass: Considering length n - 1

(**1 4** 2 5 9) ⟶ (**1 4** 2 5 9)
(1 **4 2** 5 9) ⟶ (1 **2 4** 5 9), Swap since 4 > 2
(1 2 **4 5** 9) ⟶ (1 2 **4 5** 9)

Third Pass: Considering length n -2

(**1 2** 4 5 9) ⟶ (**1 2** 4 5 9)
(1 **2 4** 5 9) ⟶ (1 **2 4** 5 9)

Fourth Pass: Considering length n -3

(**1 2** 4 5 9) ⟶ (**1 2** 4 5 9)

Example:

Rabbit Example: Array {6, 1, 2, 3, 4, 5} is almost sorted too, but it takes O(n) iterations to sort it. Element {6} is a rabbit. This example demonstrates adaptive property of the bubble sort.

CL GATEaspire AN IIT/IISc ALUMNI VENTURE

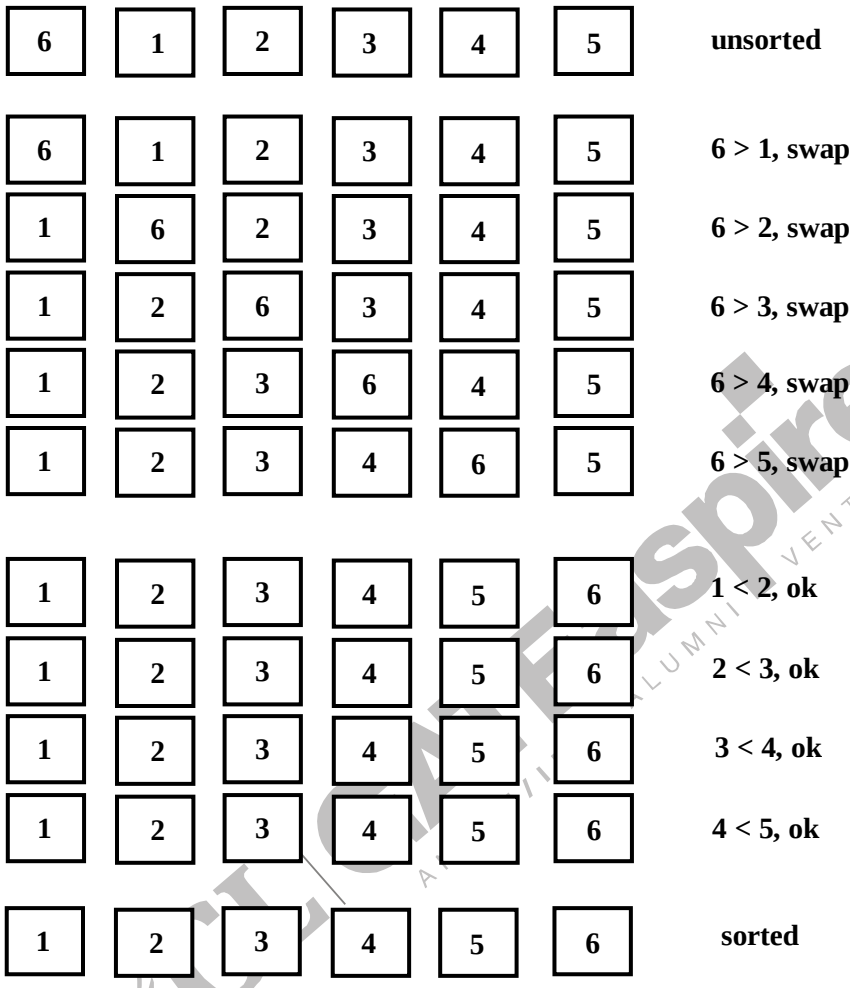

Fig. 6.1

6.1.1 Performance

The running time can be bound as the total number of comparison being made in the entire execution of this algorithm. Thus, the worst case (input is descending order) comparisons in the respective passes are as follows:

$1^{st} \longrightarrow n - 1$

$2^{nd} \longrightarrow n - 2$

…

$(n - 1)$th pass $\longrightarrow 1$

Therefore, total comparisons = $n(n-1) / 2$;

which implies $O(n^2)$ time complexity.

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. Performance of bubble sort over an already-sorted list (best-case) is $O(n)$. Remember using a flag it can easily be determined that if no swaps were performed in the current pass that would mean the list is sorted and bubble sort can come out then itself rather than going through all the remaining passes.

Remarks:

- It is comparison based sorting method.
- It is in place sorting method.
- Worst case space complexity is of O(1).
- It is adaptive sorting method as presortedness affects the running time.
- A stable sorting method.

6.2 Insertion Sort

Insertion sort is a simple a comparison sorting algorithm. Insertion sorting algorithm is similar to bubble sort. But insertion sort is more efficient than bubble sort because in insertion sort the elements comparisons are less as compare to bubble sort. Every iteration of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.

Example:

Pass 1

5 **4** 3 2 1 ⟶ **4** 5 3 2 1

Pass 2

4 5 **3** 2 1 ⟶ 4 **3** 5 2 1 ⟶ **3** 4 5 2 1

Pass 3

3 4 5 **2** 1 ⟶ 3 4 **2** 5 1⟶ 3 **2** 4 5 1 ⟶ **2** 3 4 5 1

Pass 4

2 3 4 5 **1** ⟶ 2 3 4 **1** 5 ⟶ 2 3 **1** 4 5 ⟶ 2 **1** 3 4 5 ⟶ **1** 2 3 4 5

6.2.1 Performance

The worst case input is an array sorted in reverse order. In this case every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. For this case insertion sort has a quadratic running time (i.e., $O(n^2)$). The running time can be

bound as the total number of comparison being made in the entire execution of this algorithm. Thus, the worst case comparisons in the respective passes are as follows:

$1^{st} \longrightarrow 1$
$2^{nd} \longrightarrow 2$
$3^{rd} \longrightarrow 3$
…
(n – 1)th pass ⟶ n - 1
Therefore, total comparisons = n(n -1) / 2;
which implies $O(n^2)$ time complexity.

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $\Theta(n)$). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

While insertion sort typically requires more writes because the inner loop can require shifting large sections of the sorted portion of the array. In general, insertion sort will write to the array $O(n^2)$ times, whereas selection sort will write only $O(n)$ times. For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading.

Remarks:
It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- simple implementation.
- efficient for (quite) small data sets.
- adaptive, i.e. efficient for data sets that are already substantially sorted: the time complexity is $O(n)$.
- stable, i.e. does not change the relative order of elements with equal keys.
- in-place, i.e. only requires a constant amount O(1) of additional memory space.

Most humans when ordering a deck of cards, for example—use a method that is similar to insertion sort.

6.3 Selection Sort

Selection sort is also a simple a comparison sorting algorithm. The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Selection sort is one of the $O(n^2)$ sorting algorithms, which makes it quite inefficient for sorting large data volumes. Selection sort is notable for its programming simplicity and it can over perform other sorts in certain situations (see complexity analysis for more details).

Algorithm

1. 1st iteration selects the smallest element in the array, and swaps it with the first element.
2. 2nd iteration selects the 2nd smallest element (*which is the smallest element of the remaining elements*) and swaps it with the 2nd element.
3. The algorithm continues until the last iteration selects the 2nd largest element and swaps it with the 2nd last index, leaving the largest element in the last index.

Effectively, the list is divided into two parts: the sublist of items already sorted, which is built up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements:
66 25 12 22 11
11 25 12 22 66
11 12 25 22 66
11 12 22 25 66
11 12 22 25 66

6.3.1 Performance

The all inputs are worst case input for selection sort as each current element has to be compared with the rest of unsorted array. The running time can be bound as the total number of comparison being made in the entire execution of this algorithm. Thus, the worst case comparisons in the respective passes are as follows:

$1^{st} \longrightarrow n - 1$
$2^{nd} \longrightarrow n - 2$
…
$(n - 1)$th pass $\longrightarrow 1$

Therefore total comparisons = $n(n-1) / 2$;
which implies $O(n^2)$ time complexity.

Remarks:

- It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.
- simple implementation.
- efficient for (quite) small data sets.

- not being adaptive.
- Its stability depends on the implementation of choosing minimum.
- in-place, i.e. only requires a constant amount O(1) of additional memory space.
- Insertion sort is very similar in that after the *k*th iteration, the first *k* elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs in order to place the $k + 1$st element, while selection sort must scan all remaining elements to find the $k + 1$st element.
- selection sort always performs $\Theta(n)$ swaps.

6.4 Merge Sort

Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

Merge Sort is a O(nlogn) sorting algorithm. **Merge sort** is an $O(n \log n)$ comparison-based divide and conquer sorting algorithm.

Conceptually, a merge sort works as follows:

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sublists of about half the size.
3. Sort each sublist recursively by re-applying merge sort algorithm.
4. Merge the two sublists back into one sorted list.

Example:

CL GATEaspire AN IIT/IISc ALUMNI VENTURE

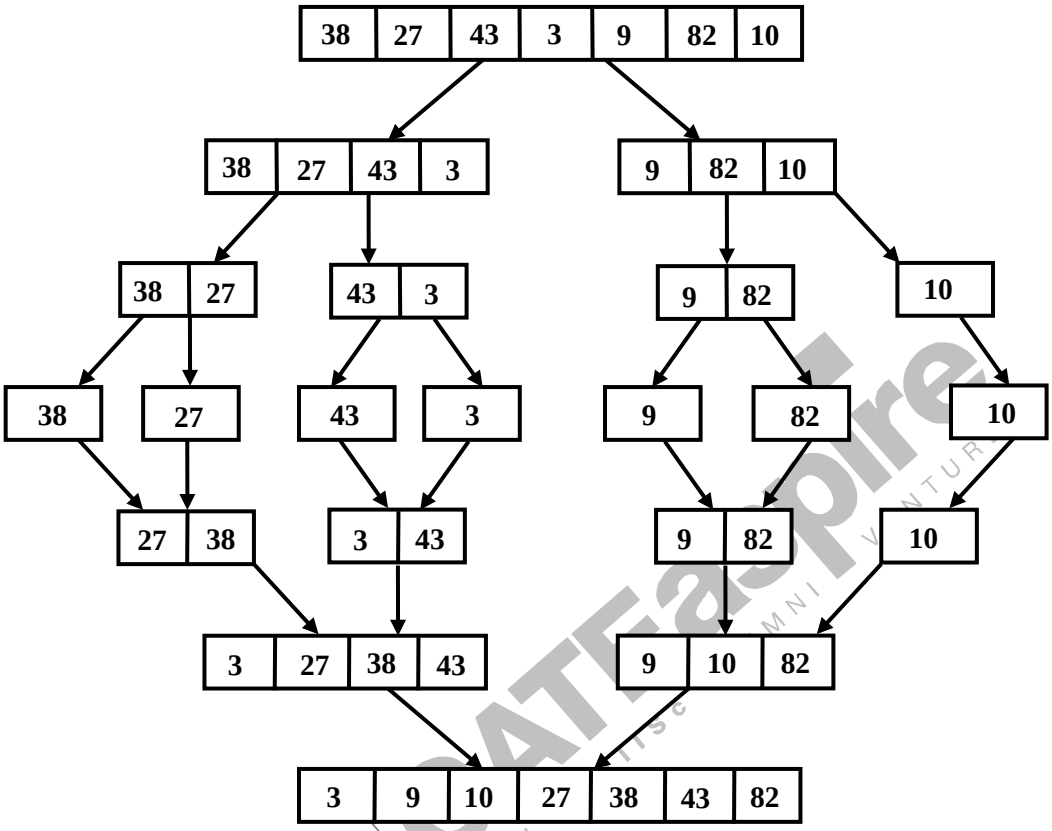

Fig. 6.2

6.4.1 Performance

In sorting *n* objects, merge sort has an average and worst-case performance of O(*n* log *n*). If the running time of merge sort for a list of length *n* is *T*(*n*), then the recurrence *T*(*n*) = 2*T*(*n*/2) + *n* follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the *n* units taken to merge the resulting two sorted lists).

Thus, after simplifying the recurrence relation;
T(n) = O(nlogn).

Remarks:

- Its not adaptive.
- merge sort is a stable sort as long as the merge operation is implemented properly.
- Not a in-place sorting method, requires O(n) of additional memory space. The additional *n* locations were needed because one couldn't reasonably merge two sorted sets in place.

6.5 Heap Sort

Heap sort is a comparison-based sorting algorithm which is much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap. Once the data list has been made into a heap, the root node is guaranteed to be the largest(or smallest) element. When it is removed (using deleteMin/deleteMax) and placed at the end of the list, the heap is rearranged so the largest element of remaining moves to the root. Using the heap, finding the next largest element takes *O(log n)* time, instead of *O(n)* for a linear scan as in simple selection sort. This allows Heapsort to run in *O(n log n)* time.

Remarks:

- Its not adaptive.
- It is in-place sorting method as utilized the same input array for placing the sorted subarray.
- Not a stable sorting method as during deleteMin/deleteMax the order is not preserved for the same key values. Consider an input that having all the same key values. The deleteMin will pickup the last heap element as to place in the root location. Thereby, the order is changed because in the sorted output later values appears before.

6.6 Quick Sort

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists. The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which are always sorted.

6.6.1 Performance

If the running time of quick sort for a list of length n is $T(n)$, then the recurrence $T(n) = T(\textit{size of } s1) + T(\textit{size of } s2) + n$ follows from the definition of the algorithm (apply the algorithm to two sets which are result of partioning, and add the n units taken by the partioning the given list).

Worst Case Time Complexity

The quick sort shows its worst case behavior when one of the subset after partioning is always empty for all subsequent recursive quick sort calls. This could have happened because of a bad choice of pivot. Then the above recurrence relation can be written as follows to represent the worst case behavior.

$T(n) = 0 + T(n-1) + n = T(n-1) + n$

Therefore, $T(n) = n + (n - 1) + (n - 2) + \ldots 1 = O(n^2)$.

Best Case Time Complexity

The quick sort shows its best case behavior when the size of each subset after partioning is always close to same for all subsequent recursive quick sort calls. This could have happened because of a good choice of pivot. Then the above recurrence relation can be written as follows to represent the best case behavior.

$T(n) = 2T(n/2) + n$

Therefore, $T(n) = O(n\log n)$.

$O(n\log n)$ is also the average case time complexity of quick sort.

Remarks:

- The known fastest in-place algorithm in practice.
- Each call finalizes the correct position of pivot element which can never be changed by any subsequent calls.
- It is an adaptive sorting method.
- Can be implemented as a stable sort depending on how the pivot is handled.

CL GATEaspire AN IIT/IISc ALUMNI VENTURE