CS241 -- Lecture Notes: Sorting Algorithm

Daisy Tang Back To Lectures Notes

This lecture covers Chapter 12 of our textbook and part of the contents are derived from Wikipedia.

Click here for the slides presentations.

Introduction to Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Sorting algorithms provide an introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, best-, worst- and average-case analysis, time-space tradeoffs, and lower bounds.

Classification:

- **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical sorting algorithms, a good behavior is $O(n \log n)$ and a bad behavior is $O(n^2)$.
- **Computational complexity of swaps** (for "in place" algorithms).
- **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only $O(1)$ or $O(\log n)$ 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.
- **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.
- General methods: **insertion**, **exchange**, **selection**, **merging**, etc.
- **Adaptability**: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Stability

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say *R* and *S*) with the same key, and *R* appears before *S* in the original list, then *R* will always appear before *S* in the sorted list. When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first component:

(4, 2) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

- (3, 7) (3, 1) (4, 2) (5, 6)
- (3, 1) (3, 7) (4, 2) (5, 6)

Additional computational cost is involved to extend the key comparison to other components.

Name	**Average**	**Worst**	**Memory**	**Stable**	**Method**
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$	No	Partitioning
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection

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. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Because it only uses comparisons to operate on elements, it is a comparison sort.

Step-by-Step Example

Assume we have an array "5 1 4 2 8" and we want to sort the array from the lowest number to the greatest number using bubble sort.

Pseudocode Implementation

```
procedure bubbleSort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length(A) - 1 inclusive do:
      if A[i] > A[i+1] then
        swap( A[i], A[i+1] )
        swapped := true
      end if
    end for
  while swapped
end procedure
```

An Improved Alternative Implementation

```
procedure bubbleSort( A : list of sortable items ) defined as:
  n := length( A )
  do
    swapped := false
    for each i in 0 to n - 1 inclusive do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
    n := n - 1
  while swapped
end procedure
```

Here, instead of doing $n(n-1)$ comparisons, we reduce it to $(n-1) + (n-2) + ... + 1 = n(n-1)/2$ comparisons.

Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

Bubble sort is not a practical sorting algorithm when n is large.

Selection Sort

Selection sort is an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

Algorithm

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)

Effectively, we divide the list into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted.

Example: Consider an example of sorting "64 25 12 22 11".

Java Implementation of the Algorithm

```
void selectionSort(int[] a) {
  for (int i = 0; i < a.length - 1; i++) {
    int min = i;
    for (int j = i + 1; j < a.length; j++) {
      if (a[j] < a[min]) {
        min = j;
      }
    }
    if (i != min) {
      int swap = a[i];
      a[i] = a[min];
      a[min] = swap;
    }
    }
}
```

Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n^2)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

How many comparisons does the algorithm need to perform? How many swaps does the algorithm perform in the worst case?

Analysis

Selecting the lowest element requires scanning all n elements (this takes n-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning all n-1 elements and so on, for (n-1) + (n-2) + ... + 2 + 1 ($O(n^2)$) comparisons. Each of these scans requires one swap for n-1 elements.

Insertion Sort

Insertion sort is a comparison sort in which the sorted array (or list) is built one entry at a time. 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 small data sets
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(n+d)$, where d is the number of inversions
- More efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort: the average running time is $n^2/4$, and the running time is linear in the best case
- 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
- Online, i.e., can sort a list as it receives it

Algorithm

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.

Sorting is typically done in-place. The resulting array after *k* iterations has the property where the first *k*+1 entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result.

Example: Consider an example of sorting "64 25 12 22 11".

Pseudocode Implementation:

```
insertionSort(array A)
begin
  for i := 1 to length[A] - 1 do
  begin
    value := A[i];
    j := i - 1;
    while j >= 0 and A[j] > value do
    begin
      A[j + 1] := A[j];
      j := j - 1;
    end;
    A[j + 1] := value;
  end;
end;
```

Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

A Comparison of the $O(n^2)$ Sorting Algorithms

Among simple average-case $O(n^2)$ algorithms, selection sort almost always outperforms bubble sort, but is generally outperformed by insertion sort.

Insertion sort's advantage is that it only scans as many elements as it needs in order to place the k+1st element, while selection sort must scan all remaining elements to find the k+1st element. Experiments show that insertion sort usually performs about half as many comparisons as selection sort. Selection sort will perform identically regardless of the order the array, while insertion sort's running time can vary considerably. Insertion sort runs much more efficiently if the array is already sorted or "close to sorted."

Selection sort always performs $O(n)$ swaps, while insertion sort performs $O(n^2)$ swaps in the average and worst case. Selection sort is preferable if writing to memory is significantly more expensive than reading.

Insertion sort or selection sort are both typically faster for small arrays (i.e., fewer than 10-20 elements). A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" subarrays.

Merge Sort

Merge sort is an $O(n \log n)$ comparison-based sorting algorithm. It is an example of the divide and conquer algorithmic paradigm.

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.
4. Merge the two sublists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted.

```
Algorithm MergeSort(A, 0, n-1)
  MergeSort(A, 0, n/2)
  MergeSort(A, n/2 + 1, n-1)
  MergeTogether(2 arrays above)
```

Example: sort the following numbers 35, 62, 33, 20, 5, 72, 48, 50.

Performance

- Worst case performance: $O(n \log n)$
- Best case performance: $O(n \log n)$ typical
- Average case peroformance: $O(n \log n)$
- Worst case space complexity: $O(n)$ total, $O(n)$ auxiliary

Analysis

Let's analyze the big-Oh of the above MergeSort.

We can solve the recurrence relation given above. We'll write n instead of *O(n)* in the first line below because it makes the algebra much simpler.

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

     =  2 [2 T(n/4) + n/2] + n

     =  4 T(n/4) + 2n

      = 4 [2 T(n/8) + n/4] + 2n

      = 8 T(n/8) + 3n

      = (fill in this line)

       = ......

      = 2^k T(n/2^k) + k n
```

We know that `T(1) = 1` and this is a way to end the derivation above. In particular we want `T(1)` to appear on the right hand side of the = sign. This means we want:

$$n/2^k = 1 \quad OR \quad n = 2^k \quad OR \quad \log_2 n = k$$

Continuing with the previous derivation we get the following since $k = \log_2 n$:

$$= 2^k\ T(n/2^k) + k\ n$$

```
= 2^(log2 n) T(1) + (log2n) n

= n + n log2 n    [remember that T(1) = 1]

= O(n log n)
```

So we've solved the recurrence relation and its solution is what we "knew" it would be. To make this a formal proof you would need to use induction to show that O(*n* log *n*) is the solution to the given recurrence relation, but the "plug and chug" method shown above shows how to derive the solution --- the subsequent verification that this is the solution is something that can be left to a more advanced algorithms class.

Quicksort

Quicksort is a well-known sorting algorithm that, on average, makes $O(n \log n)$ comparisons to sort n items. However, in the worst case, it makes $O(n^2)$ comparisons. Typically, quicksort is significantly faster than other $O(n \log n)$ algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data, it is possible to make design choices which minimize the probability of requiring quadratic time.

Quicksort is a comparison sort and, in efficient implementations, is not a stable sort.

Algorithm

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called the *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.

```
function quicksort(array)
    var list less, greater
    if length(array) <= 1
      return array
    select and remove a pivot from array
    for each x in array
      if x <= pivot then append x to less
      else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

Quicksort is similar to merge sort in many ways. It divides the elements to be sorted into two groups, sorts the two groups by recursive calls, and combines the two sorted groups into a single array of sorted values. However, the method for dividing the array in half is much more sophisticated than the simple method we used for merge sort. On the other hand, the method for combining these two groups of sorted elements is trivial compared to the method used in mergesort.

The correctness of the partition algorithm is based on the following two arguments:

- At each iteration, all the elements processed so far are in the desired position: before the pivot if less than or equal to the pivot's value, after the pivot otherwise.
- Each iteration leaves one fewer element to be processed.

The disadvantage of the simple version above is that it requires $O(n)$ extra storage space, which is as bad as merge sort. The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complex version which uses an in-place partition algorithm and use much less space.

The partition pseudocode:

```
private static int partition(int[] data, int first, int n) {
  1. Initialize values:
       pivot = data[first];
       tooBigIndex = first + 1;         // Index of element after pivot
       tooSmallIndex = first + n - 1;   // Index of last element

  2. Repeat the following until the two indices cross each other
     2a. while tooBigIndex is not yet beyond the final index of the part
of the array we are partitioning,
         and data[tooBigIndex] is less than or equal to the pivot, move
tooBigIndex to the next index.
```

```
     2b. while data[tooSmallIndex] is greater than the pivot, move
tooSmallIndex down to the previous index.

     2c. if (tooBigIndex < tooSmallIndex), swap the values of
data[tooBigIndex] and data[tooSmallIndex].

  3. Move the pivot element to its correct position at
data[tooSmallIndex], return tooSmallIndex
```

Choosing a Good Pivot Element

The choice of a good pivot element is critical to the efficiency of the quicksort algorithm. If we can ensure that the pivot element is near the median of the array values, then quicksort is very efficient. One technique that is often used to increase the likelihood of choosing a good pivot element is to randomly choose three values from the array and then use the middle of these three values as the pivot element.

Let's try the quicksort algorithm with the following array: 40, 20, 10, 80, 60, 50, 7, 30, 100, 90, and 70.

Heapsort

Heapsort is a comparison-based sorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of quicksort, it has the advantage of a worst-case $O(n \log n)$ runtime. Heapsort combines the time efficiency of merge sort and the storage efficiency of quicksort.

Heapsort is similar to selection sort in that it locates the largest value and places it in the final array position. Then it locates the next largest value and places it in the next-to-last array position and so forth. However, heapsort uses a much more efficient algorithm to locate the array values to be moved.

How it works?

The heapsort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array.

This is repeated until there are no items left in the heap and sorted array is full. Elementary implementations require two arrays -- one to hold the heap and the other to hold the sorted elements. In advanced implementation however, we have an efficient method for representing a heap (complete binary tree) in an array and thus do not need an extra data structure to hold the heap.

Let's try the algorithm with the following binary heap stored in an array: 45, 27, 42, 21, 23, 22, 35, 19, 4, and 5.

Last updated: Nov. 2012