Sorting algorithms

Properties of sorting algorithm

1) **Adaptive**: speeds up to O(n) when data is nearly sorted
2) **Stable**: does not change the relative order of elements with equal keys
3) **In**-**place**: only requires a constant amount O(1) of additional memory space
4) **Online**: can sort a list as it receives it

Bubble sort

Sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps 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. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.
The bubble sort algorithm can be easily optimized by observing that the n^{th} pass finds the n^{th} largest element and puts it into its final place. So, the inner loop can avoid looking at the last n-1 items when running for the n^{th} time.

Algorithm:

```
for i = 1:n,
   swapped = false
   for j = n:i+1,
      if a[j] < a[j-1],
         swap a[j,j-1]
         swapped = true
   break if not swapped
end
```

Properties:

- Stable
- In-place
- $O(n^2)$ comparisons and swaps
- Not online
- Adaptive: O(n) when nearly sorted

Example

Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.
Given data: 6 5 3 1 8

Pass 1:
6 5 3 1 8
5 6 3 1 8
5 3 6 1 8
5 3 1 6 8
5 3 1 6 8

Pass 2:

$\underline{5\ 3}$ 1 6 8

3 $\underline{5\ 1}$ 6 8

3 1 $\underline{5\ 6}$ 8

3 1 5 6 8

Pass 3:

$\underline{3\ 1}$ 5 6 8

1 $\underline{3\ 5}$ 6 8

1 3 5 6 8

Pass 4:

$\underline{1\ 3}$ 5 6 8

1 3 5 6 8

Insertion sort

It is a simple sorting algorithm that builds the final sorted array (or list) one item 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 such as Simple implementation, efficient for (quite) small data sets, more efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort and usually faster in practice than asymptotically faster algorithms for small data sets.

Algorithm:

```
for i = 2:n,
   for (k = i; k > 1 and a[k] < a[k-1]; k--)
      swap a[k], a[k-1]
end
```

Properties:

- Stable
- In-place
- $O(n^2)$ comparisons and swaps
- Online
- Adaptive: O(n) when nearly sorted
- Very less overhead

Example

Given data: 5 6 3 1 8

Pass 1:
5 6 3 1 8 //compare 6 with all its previous and swap if previous are greater

Pass 2:
5 6 3 1 8
5 3 6 1 8
3 5 6 1 8

Pass 3:
3 5 6 1 8
3 5 1 6 8
3 1 5 6 8
1 3 5 6 8

Pass 4:
1 3 5 6 8
1 3 5 6 8

Shell sort

It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort). The worse-case time complexity of shell sort depends on the increment sequence. The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements can move some out-of-place elements into position faster than a simple nearest neighbor exchange.

Algorithm:

```
gaps = [x, y, z, …]
for each (G in gaps)
        create sub-arrays with elements having gap as G
        for each(S in sub-arrays)
                insertion_sort(s)
        merge all sub-arrays
```

Properties:

- Not Stable
- In-place
- Complexity depends upon gap
- Not online
- Adaptive: O(n.log(n)) when nearly sorted

Example

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
input data:	62	83	18	53	07	17	95	86	47	69	25	28
after 5-sorting:	17	28	18	47	07	25	83	86	53	69	62	95
after 3-sorting:	17	07	18	47	28	25	69	62	53	83	86	95
after 1-sorting:	07	17	18	25	28	47	53	62	69	83	86	95

The first pass, 5-sorting, performs insertion sort on separate subarrays (a1, a6, a11), (a2, a7, a12), (a3, a8), (a4, a9), (a5, a10). For instance, it changes the subarray (a1, a6, a11) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on the subarrays (a1, a4, a7, a10), (a2, a5, a8, a11), (a3, a6, a9, a12). The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a1,..., a12).

Heap sort

heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a heap data structure. Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime.

Algorithm:

1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the shiftDown() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

Properties:

- Not stable
- In-place
- O(n log n) comparisons and swaps
- Not online
- Not really adaptive

Example

Refer Heap tree notes

Quick sort

It is a simple sorting algorithm that builds the final sorted array (or list) one item 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 such as Simple implementation, efficient for (quite) small data sets, more efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort and usually faster in practice than asymptotically faster algorithms for small data sets.

The steps are:

1. Pick an element, called a pivot, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values 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 apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Algorithm:

```
quicksort(A, lo, hi)
  if (lo < hi)
    p = partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)

partition(A, lo, hi)
    pivot = A[lo]
    i = lo + 1
    j = hi
    while True
        do        j--      while A[j] > pivot
        do        i++      while A[i] < pivot
        if (i < j)
            swap A[i] with A[j]
        else
            return j
```

Properties:

- Not stable
- In-place
- O(n log n) comparisons and swaps
- Not online
- Adaptive
- less overhead

Example

Refer class notes

Bucket sort

It works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort. Bucket sort is a generalization of pigeonhole sort. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.

Algorithm:

```
bucketSort(array, n)
        buckets ← new array of n empty lists
        for i = 0 to (length(array)-1)
                insert array[i] into buckets[msbits(array[i], k)]
        for i = 0 to n - 1 do
                sort(buckets[i]);
        return the concatenation of buckets[0], ...., buckets[n-1]
```

Properties:

- Stable
- Not in-place
- Not really online
- Not adaptive
- Overhead of buckets

Example

Given data: 29 25 3 49 9 37 21 43

//inserting data in buckets
Bucket (0-9): 3 9
Bucket (10-19):
Bucket (20-29): 29 25 21
Bucket (30-39): 37
Bucket (40-49): 49 43

//sorting individual bucket
Bucket (0-9): 3 9
Bucket (10-19):
Bucket (20-29): 21 25 29
Bucket (30-39): 37
Bucket (40-49): 43 49

//merge all buckets
3 9 21 25 29 37 43 49

Merge sort

It is a simple sorting algorithm that builds the final sorted array (or list) one item 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 such as Simple implementation, efficient for (quite) small data sets, more efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort and usually faster in practice than asymptotically faster algorithms for small data sets.

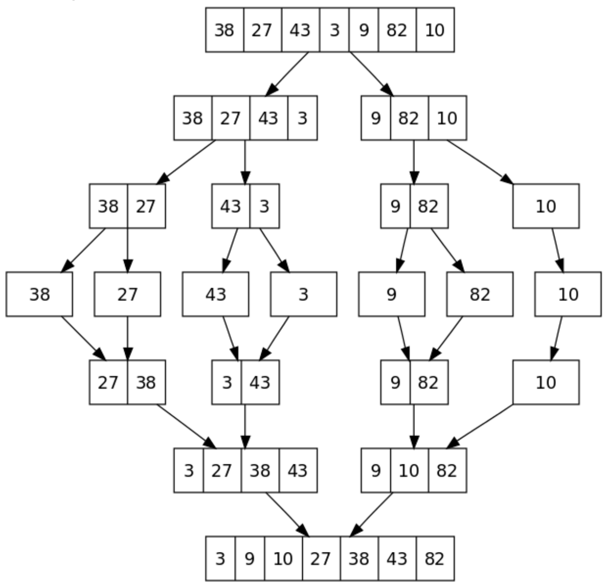
Algorithm:

```
merge_sort(m)
        if(m==1)
                return
        else
                middle = length(m)/2
                left_list = (1 to middle) elements of m
                right_list = (middle+1 to length(m)) elements of m
                merge(left_list, right_list)

merge(left, right)
        while !empty(left) and !empty(right)
                if first(left) <= first(right)
                        append first(left) to result
                        left = rest(left)
                else
                        append first(right) to result
                        right = rest(right)

        while !empty(left)
                append left to result

        while !empty(right)
                append right to result
        return result
```

Properties:

- Stable
- In-place
- Not online
- Not adaptive
- Overhead for linked list and extra arrays

Example


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Radix sort

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers.

LSD radix sorts process the integer representations starting from the least digit and move towards the most significant digit. LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11. MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j".

Algorithm:

1. Take the least significant digit (or group of bits, both being examples of radices) of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys. (This is what makes the LSD radix sort a stable sort.)
3. Repeat the grouping process with each more significant digit.

Properties:

- Stable
- Not in-place
- Not online
- Adaptive

Example

Original, unsorted list:
170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:
170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:
802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:
2, 24, 45, 66, 75, 90, 170, 802

Complexity of sorting algorithms

Name	**Best**	**Average**	**Worst**	**Memory**	**Stable**	**Method**
Quick sort	n log n	n log n	n^2	n	No	Partitioning
Merge sort	n log n	n log n	n log n	n	Yes	Merging
Heap sort	n log n	n log n	n log n	1	No	Selection
Insertion sort	n	n^2	n^2	1	Yes	Insertion
Selection sort	n^2	n^2	n^2	1	No	Selection
Shell sort	n	$n \log^2 n$	Depends on gap sequence	1	No	Insertion
Bubble sort	n	n^2	n^2	1	Yes	Exchanging
LSD radix sort	-	n*k/d	n*k/d	$n+2^d$	Yes	Non comparison sort
Bucket sort	-	n+r	n+r	n+r	Yes	
k key size, d digit size, r range of numbers						

File structure

- A file is collection of records
- Record is information stored in rows
- Every row is dedicated to particular entity
- Every row is made up of fields
- Every field holds the value for each piece of information topic (column)
- Every row have one key, a field with unique value, used to identify that row, also called as primary key

Operations on file

- **File pointer declaration**
  file *fp;
- **Creating/ opening a file**
  fp = fopen("temp.dat","w+");//open temp.dat file in W+ mode
  Creates a file and then open same if not exist
  file handling modes
  r read mode
  r+ read and write existing file
  w opens file in write mode
  w+ read, write and modify existing contents
  a append data
  a+ read but cant modify previous data, append new
- **Writing a character to a file**
  char ch='x';
  fputc(ch,fp);
- **Writing a string to a file**
  char s[]=”hello”;
  fputs(s,fp);
- **Writing variable value to a file**
  int i=10;
  char ch=’x’;
  float f=10.35;
  char s[]=”hi”;
  fprintf(fp, “%d %c %f %s”, &I, &ch, &f, &s);
- **Set file pointer to start of file**
  rewind(fp);
- **Reading a character from a file**
  char ch = fgets(fp);
- **Reading a string from a file**
  char s[10];
  fgets(s,3,fp); //read 2 character from a fp store in s
- **Closing a file**
  fclose(fp);

Types of file

1. **Sequential file**
    - Data are stored as they are read, writing is faster
    - Simple to manage
    - Hence unsorted data
    - Takes time to find data and read it
    - Not good choice for large data, less efficient
    - Ex.

      Data as follows

Roll no	Name	Age
1	ABC	23
7	CDE	27
3	XYZ	21

Data stored is same as above

2. **Index Sequential file**
    - Maintains two files i.e. normal sequential file and sorted index file
    - All data are stored in sequential file as they are read
    - Primary key and offset(address of storage location) is stored in index file
    - As new data is added index file gets sorted
    - To search a record, search primary key of record in index file, if matched take address of record and read from sequential file
    - It is more efficient than sequential file as data can be quickly accessed
    - It requires additional storage apart for data
    - Requires index file to get sorted as new data is added
    - Good choice where data is larger
    - Ex. Data as follows

Roll no	Name	Age
1	ABC	23
7	CDE	27
3	XYZ	21

Then sequential file as follows

Offset	Roll no	Name	Age
1000	1	ABC	23
1500	7	CDE	27
1750	3	XYZ	21

and index file as follows

Roll no	Offset
1	1000
7	1500
3	1750