

DATA STRUCTURES

Sudent name:- Shaima AbuNaim ;

Student NO:- 1110885;

Instructor:- Iyad Jaber; .

Section:-2;

Date:- 15/5/2013;

## Contents

1. Introduction.
2. Bubble Sort.
3. Insertion Sort.
4. Selection Sort.
5. Radix Sort.
6. Shell Sort.
7. Merge Sort.
8. Heap Sort.
9. Tim Sort.
10. Quick Sort.
11. Counting Sort.
12. Cocktail Sort.
13. Comb Sort.
14. Conclusion.

Introduction:

Sorting

One of the most common applications in computer science is sorting, through which data

are arranged according to their values.

Let A be a list of n elements A1,A2……………AN in memory. Sorting of A means the

operation of rearranging the contents of A so that they are increasing in order

(numerically or lexicographically), so that

A1 <=A 2< =A3 < = A4…………<=AN.

Since A has n elements, there are n! Ways that the contents can appear in A. these ways

correspond to the n! Permutation of 1, 2, 3……..n. accordingly, each sorting algorithm

must take care of these n! Possibilities.

***Bubble Sort:***

**Bubble Sort:** - The simplest sorting algorithm. It takes two array elements at a time, compares them and swaps their positions if element on left is greater than right. Divides the array elements in two halves or partitions. On dividing, the quick sort procedure is recursively called to sort the two halves. A “pivot” is used as the center point and elements less than the pivot are moved to the left or before the pivot and elements greater than pivot are moved to the right.

Abubble Sort Example:

|  |
| --- |
|  6 |
| 5 |
| 4 |
| 3 |
| 2 |
| 1 |

We start by comparing the first

two elements in the List.

Compare

This list is an example of a

“worst case” scenario for sorting,

because the List is in the exact

opposite order from the way we

want it sorted (the computer

program does not know this).

|  |
| --- |
| 5 |
| 6 |
| 4 |
| 3 |
| 2 |
| 1 |

|  |
| --- |
| 5 |
| 6 |
| 4 |
| 3 |
| 2 |
| 1 |

|  |
| --- |
| 5 |
| 4 |
| 6 |
| 3 |
| 2 |
| 1 |

Swap

Swap

Compare

|  |
| --- |
| 5 |
| 4 |
| 6 |
| 3 |
| 2 |
| 1 |

|  |
| --- |
| 5 |
| 4 |
| 3 |
| 6Compare |
| 2 |
| 1 |

|  |
| --- |
| 5 |
| 4 |
| 3 |
| 6 |
| 2 |
| 1 |

Swap

Compare

|  |
| --- |
| 5 |
| 4 |
| 3 |
| 2 |
| 6 |
| 1 |

|  |
| --- |
| 5 |
| 4 |
| 3 |
| 2 |
| 6 |
| 1 |

Swap

Compare

As you can see, the largest

|  |
| --- |
| 5 |
| 4 |
| 3 |
| 2 |
| 6 |
| 1 |

number has “bubbled” down to

the bottom of the List after the

first pass through the List.

Swap

|  |
| --- |
| 5 |
| 4 |
| 3 |
| 2 |
| 1 |
| 6 |

Compare

For our second pass through

the List, we start by

comparing these first two

elements in the List.

|  |
| --- |
| 4 |
| 5compare |
| 3 |
| 2 |
| 1 |
| 6 |

|  |
| --- |
| 4 |
| 5 |
| 3 |
| 2 |
| 1 |
| 6 |

|  |
| --- |
| 4 |
| 3 |
| 5 |
| 2 |
| 1 |
| 6 |

Swap

Swap

|  |
| --- |
| 4 |
| 3 |
| 5 |
| 2 |
| 1 |
| 6 |

|  |
| --- |
| 4 |
| 3 |
| 2 |
| 5 |
| 1 |
| 6 |

|  |
| --- |
| 4 |
| 3 |
| 2 |
| 5 |
| 1 |
| 6 |

compare

Swap

compare

|  |
| --- |
| 4 |
| 3 |
| 2 |
| 1 |
| 5 |
| 6 |

At the end of the second pass, we

stop at element number n - 1,

because the largest element in the

List is already in the last position.

Swap

Swap

|  |
| --- |
| 4 |
| 3 |
| 2 |
| 1 |
| 5 |
| 6 |

We start with the first two

compare

elements again at the beginning

of the third pass.

|  |
| --- |
| 3 |
| 4compare |
| 2 |
| 1 |
| 5 |
| 6 |

|  |
| --- |
| 3 |
| 4 |
| 2 |
| 1 |
| 5 |
| 6 |

|  |
| --- |
| 3 |
| 2 |
| 4 |
| 1 |
| 5 |
| 6 |

Swap

Swap

|  |
| --- |
| 3 |
| 2 |
| 4 |
| 1 |
| 5 |
| 6 |

|  |
| --- |
| 3 |
| 2 |
| 1 |
| 4 |
| 5 |
| 6 |

compare

At the end of the third pass, we stop

comparing and swapping

Swap

 at element number n - 2.

|  |
| --- |
| 3 |
| 2 |
| 1 |
| 4 |
| 5 |
| 6 |

The beginning of the fourth pass...

compare

|  |
| --- |
| 2 |
| 3 |
| 1 |
| 4 |
| 5 |
| 6 |

|  |
| --- |
| 2 |
| 3 |
| 1 |
| 4 |
| 5 |
| 6 |

Swap

compare

|  |
| --- |
| 2 |
| 1 |
| 3 |
| 4 |
| 5 |
| 6 |

The end of the fourth pass

Swap

stops at element number n - 3.

|  |
| --- |
| 2 |
| 1 |
| 3 |
| 4 |
| 5 |
| 6 |

compare

The beginning of the fifth pass...

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

The last pass compares only

Swap

the first two elements of the

List. After this comparison

and possible swap, the

smallest element has

“bubbled” to the top.

***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

 *→ invariant: a[1..i] in final position*

 break if not swapped

end

***Advantage and Disadvantage of Bubble Sort:***

Advantage is simplicity.

Disadvantage is that it can take N scans, where N is the size of the array or list, because an out of position item is only moved one position per scan. This can be mitigated somewhat by starting with a swap gap of greater than one (typically N/2), scanning until no swaps occur, then halving the gap and repeating until the gap is one. This, of course, is no longer a bubble sort - it is a merge exchange sort.

***Performance of Bubble Sort***

Bubble sort has a worst-case and average complexity of O(n2), where n is the number of items sorted. Unlike the other sorting algorithms, bubble sort detects whether the sorted list is efficiently built into the algorithm. Bubble sort performance over an already sorted list is O(n).

The position of elements in bubble sort plays an important role in determining performance. Large elements at the beginning do not pose a problem as they are easily swapped. The small elements toward the end move to the beginning slowly. As such, these elements are called rabbits and turtles.

The bubble sort algorithm can be optimized by placing larger elements in the final position. After every pass, all elements after the last swap are sorted and do not need to be checked again, thereby skipping the tracking of swapped variables

### *Insertion sort*

*Insertion sort* is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space

## *Algorithm*

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



becomes



with each element greater than *x* copied to the right as it is compared against *x*.

The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called *Insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform an insertion sort, begin at the left-most element of the array and invoke *Insert* to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

[Pseudocode](http://en.wikipedia.org/wiki/Pseudocode) of the complete algorithm follows, where the arrays are [zero-based](http://en.wikipedia.org/wiki/Zero-based_numbering):

 for i ← 1 to i ← length(A)-1

 {

 *//The values in A[ i ] are checked in-order, starting at the second one*

 *// save A[i] to make a hole that will move as elements are shifted*

 *// the value being checked will be inserted into the hole's final position*

 valueToInsert ← A[i]

 holePos ← i

 *// keep moving the hole down until the value being checked is larger than*

 *// what's just below the hole <!-- until A[holePos - 1] is <= item -->*

 while holePos > 0 and valueToInsert < A[holePos - 1]

 { *//value to insert doesn't belong where the hole currently is, so shift*

 A[holePos] ← A[holePos - 1] *//shift the larger value up*

 holePos ← holePos - 1 *//move the hole position down*

 }

 *// hole is in the right position, so put value being checked into the hole*

 A[holePos] ← valueToInsert

 }

Note that although the common practice is to implement in-place, which requires checking the elements in-order, the order of checking (and removing) input elements is actually arbitrary. The choice can be made using almost any pattern, as long as all input elements are eventually checked (and removed from the input).

## Best, worst, and average cases

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., [Θ](http://en.wikipedia.org/wiki/Big_Theta_notation)(*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.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(*n*2)).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than [quicksort](http://en.wikipedia.org/wiki/Quicksort); indeed, good [quicksort](http://en.wikipedia.org/wiki/Quicksort) implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the item under consideration is underlined. The item that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 5 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 6 7 9 1

1 2 3 4 5 6 7 9

***Advantage and disadvantage of insertion sort:***

• Advantages

\* Best performance is best possible

\* Works well with partially sorted lists

\* Stable sort

\* Simple to understand and code (maybe)

• Disadvantages

\* Poor performance in random order case

\* O(N2) inappropriate for large arrays

***Selection Sort:***

Selection sort works by finding the smallest unsorted item in the list and swapping it with the item in the current position.

For example, consider the following array, shown with array elements in sequence separated by commas:



The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in bold. We then swap the element at index 2 with that at index 4. The result is:



We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in bold):



The next two steps give us:



Algorithm

Pseudocode

Selection-Sort(A)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|

|  |  |
| --- | --- |
|  | 1 for i ← 1 to length[A] - 1 |

|  |  |
| --- | --- |
|  | 2 do min ← i |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 3 |  |       for j ← i + 1 to length[A] |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 4 |  |       do if A[j] < A[min] |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 5 |  |       |  |          then min = j |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 6 |  |       swap A[i] with A[min] |

 |

Assume the list is to be sorted in ascending order. The idea of selection sort is as simple as follow:

* Linear-search for the minimum remaining element in the unsorted portion of the list.
* Exchange with the element of the corresponding position in sorting order. That is,
* Repeat those two steps for the rest of the list.

![A [1 ...i- 1]         A[i]         A [i+ 1...n] ◟----◝◜----◞         ◟◝◜◞         ◟----◝◜----◞    sorted list min element from A [i] unsorted list ]()

***Time Complexity***

In the first round, it requires scanning all n elements, taking n- 1 comparisons, and swapping it into the first position. Next, finding the next lowest element requires scanning the remaining n - 1 elements and making n - 2 comparisons and perform the swap. Thus, a total T(n) = (n - 1) + (n - 2) + ... + 2 + 1 =  = Θ(n2) of comparisons and n swaps have been done after the list is sorted. Thus, the selection sort always performs Θ(n2) of running time. However, one disadvantage is that the running time of selection sort barely depends on the amount of order already in file. That is, it takes as long running time to run selection sort on a list that is already sorted, or a list of duplicated keys, as it takes for a list of distinct random values.

***Space Complexity***

This is an in-place algorithm, which means no more than constant space is required to perform the operation. Thus, the space complexity is O(1).

***Stability***

Selection sort is not stable, because the order of occurrence of the same key is hard to maintain after swap. However, there is a way to implement selection sort into a stable sort: replacing swap with insertion operation. That is, all other elements moved down by one index after insertion is performed. Then, the algorithm is stable, but with greater overhead of Θ(n2) writes to memory. This change eliminates the main advantage of selection sort over insertion sort, which is always stable.

***Advantage and Disadvantage of Selection Sort:***

The main advantage of the selection sort is that it performs well on a small list. Furthermore, because it is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list. The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items. Similar to the bubble sort, the selection sort requires n-squared number of steps for sorting n elements. Additionally, its performance is easily influenced by the initial ordering of the items before the sorting process. Because of this, the selection sort is only suitable for a list of few elements that are in random order.

***Radix Sort:***

**Definition:** A multiple pass [distribution sort](http://xlinux.nist.gov/dads/HTML/distributionSort.html) algorithm that distributes each item to a [bucket](http://xlinux.nist.gov/dads/HTML/bucket.html) according to part of the item's [key](http://xlinux.nist.gov/dads/HTML/key.html) beginning with the least significant part of the key. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key.

For example Consider the following 9 numbers:

493   812   715   710   195   437   582   340   385

We should start sorting by comparing and ordering the one's digits:

|  |  |
| --- | --- |
| Digit | Sublist |
| 0 |   340 710 |
| 1 |   |
| 2 |   812 582 |
| 3 |   493 |
| 4 |   |
| 5 |   715 195 385 |
| 6 |   |
| 7 |   437 |
| 8 |   |
| 9 |   |

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340   710   812   582   493   715   195   385   437

Note: The order in which we divide and reassemble the list is extremely important, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the ten's digit:

|  |  |
| --- | --- |
| Digit | Sublist |
| 0 |   |
| 1 |   710 812 715 |
| 2 |   |
| 3 |   437 |
| 4 |   340 |
| 5 |   |
| 6 |   |
| 7 |   |
| 8 |   582 385 |
| 9 |   493 195 |

Now the sublists are gathered in order from 0 to 9:

710   812   715   437   340   582   385   493   195

Finally, the sublists are created according to the hundred's digit:

|  |  |
| --- | --- |
| Digit | Sublist |
| 0 |   |
| 1 |   195 |
| 2 |   |
| 3 |   340 385 |
| 4 |   437 493 |
| 5 |   582 |
| 6 |   |
| 7 |   710 715 |
| 8 |   812 |
| 9 |   |

At last, the list is gathered up again:

195   340   385   437   493   582   710   715   812

***Algorithm for radix sorting:***

1. Look at the rightmost digit.

2. Assign the full number to that digits index.

3. Look at the next digit to the left FROM the current

 sorted array. IF there is no digit, pad a 0.

4. REPEAT STEP 3 UNTIL all numbers have been sorted.

## *Pseudocode*

function bucketSort(array, n) is

 buckets ← new array of n empty lists

 for i = 0 to (length(array)-1) do

 insert *array[i]* into buckets[msbits(array[i], k)]

 for i = 0 to n - 1 do

 nextSort(buckets[i])

 return the concatenation of buckets[0], ...., buckets[n-1]

|  |  |
| --- | --- |
| Class | [Sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) |
| Data structure | [Array](https://en.wikipedia.org/wiki/Array_data_structure) |
| [Worst case performance](https://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [Average case performance](https://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n+k) |
| [Worst case space complexity](https://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n\cdot k) |

***Performance of Radix Sort:***

***Advantage and Disadvantage for Radix Sort:***

Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact one of the fastest sorting algorithms for numbers or strings of letters.

### Disadvantages

Still, there are some tradeoffs for Radix Sort that can make it less preferable than other sorts.

The speed of Radix Sort largely depends on the inner basic operations, and if the operations are not efficient enough, Radix Sort *can* be slower than some other algorithms such as Quick Sort and Merge Sort. These operations include the insert and delete functions of the sublists and the process of isolating the digit you want.

In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.

Radix Sort can also take up more space than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sublist for each of the possible digits or letters. If you are sorting pure English words, you will need at least 26 different sublists, and if you are sorting alphanumeric words or sentences, you will probably need more than 40 sublists in all!

Since Radix Sort depends on the digits or letters, Radix Sort is also *much* less flexible than other sorts. For every different type of data, Radix Sort needs to be rewritten, and if the sorting order changes, the sort needs to be rewritten again. In short, Radix Sort takes more time to write, and it is very difficult to write a general purpose Radix Sort that can handle all kinds of data.

***Shell Sort:***

*Shell sort* was invented by [Donald Shell](http://en.wikipedia.org/wiki/Donald_Shell) in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

## *Algorithm*

h = 1

while h < n, h = 3\*h + 1

while h > 0,

 h = h / 3

 for k = 1:h, insertion sort a[k:h:n]

 *→ invariant: each h-sub-array is sorted*

end

## *Properties*

* Not stable
* O(1) extra space
* O(n3/2) time as shown (see below)
* Adaptive: O(n·lg(n)) time when nearly sorted

## *Discussion*

The worse-case time complexity of shell sort depends on the increment sequence. For the increments *1 4 13 40 121...*, which is what is used here, the time complexity is O(n3/2). For other increments, time complexity is known to be O(n4/3) and even O(n·lg2(n)). Neither tight upper bounds on time complexity nor the best increment sequence are known.

Because shell sort is based on insertion sort, shell sort inherits insertion sort's adaptive properties. The adapation is not as dramatic because shell sort requires one pass through the data for each increment, but it is significant. For the increment sequence shown above, there are log3(n) increments, so the time complexity for nearly sorted data is O(n·log3(n)).

Because of its low overhead, relatively simple implementation, adaptive properties, and sub-quadratic time complexity, shell sort may be a viable alternative to the O(n·lg(n)) sorting algorithms for some applications when the data to be sorted is not very large.

***Merge Sort:***

*Merge sort* takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(*n* log *n*). Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages [Perl](http://en.wikipedia.org/wiki/Perl),[[7]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-6) [Python](http://en.wikipedia.org/wiki/Python_%28programming_language%29) (as [timsort](http://en.wikipedia.org/wiki/Timsort)[[8]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-7)), and [Java](http://en.wikipedia.org/wiki/Java_%28programming_language%29) (also uses timsort as of [JDK7](http://en.wikipedia.org/wiki/JDK7)[[9]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-8)), among others. Merge sort has been used in Java at least since 2000 in JDK1.3.[[10]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-mergesort_in_jdk13-9)[[11]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-jdk13_since_2000-10)

## *Algorithm*

*# split in half*

m = n / 2

*# recursive sorts*

sort a[1..m]

sort a[m+1..n]

*# merge sorted sub-arrays using temp array*

b = copy of a[1..m]

i = 1, j = m+1, k = 1

while i <= m and j <= n,

 a[k++] = (a[j] < b[i]) ? a[j++] : b[i++]

 *→ invariant: a[1..k] in final position*

while i <= m,

 a[k++] = b[i++]

 *→ invariant: a[1..k] in final position*

## *Properties*

* Stable
* Θ(n) extra space for arrays (as shown)
* Θ(lg(n)) extra space for linked lists
* Θ(n·lg(n)) time
* Not adaptive
* Does not require random access to data

## *Discussion*

Merge sort is very predictable. It makes between 0.5\*lg(n) and lg(n) comparisons per element, and between lg(n) and 1.5\*lg(n) swaps per element. The minima are achieved for already sorted data; the maxima are achieved, on average, for random data. If using Θ(n) extra space is of no concern, then merge sort is an excellent choice: It is simple to implement, and it is the only stable O(n·lg(n)) sorting algorithm. Note that when sorting linked lists, merge sort requires only Θ(lg(n)) extra space (for recursion).

Merge sort is the algorithm of choice for a variety of situations: when stability is required, when sorting linked lists, and when random access is much more expensive than sequential access (for example, external sorting on tape).

There do exist linear time *in-place* merge algorithms for the last step of the algorithm, but they are both expensive and complex. The complexity is justified for applications such as external sorting when Θ(n) extra space is not available.

### *Heapsort*

*Heapsort* is a much more efficient version of [selection sort](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Heap_%28data_structure%29), a special type of [binary tree](http://en.wikipedia.org/wiki/Binary_tree). 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 and placed at the end of the list, the heap is rearranged so the largest element 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, and this is also the worst case complexity.

## *Algorithm*

*# heapify*

for i = n/2:1, sink(a,i,n)

*→ invariant: a[1,n] in heap order*

*# sortdown*

for i = 1:n,

 swap a[1,n-i+1]

 sink(a,1,n-i)

 *→ invariant: a[n-i+1,n] in final position*

end

*# sink from i in a[1..n]*

function sink(a,i,n):

 *# {lc,rc,mc} = {left,right,max} child index*

 lc = 2\*i

 if lc > n, return *# no children*

 rc = lc + 1

 mc = (rc > n) ? lc : (a[lc] > a[rc]) ? lc : rc

 if a[i] >= a[mc], return *# heap ordered*

 swap a[i,mc]

 sink(a,mc,n)

## *Properties*

* Not stable
* O(1) extra space (see discussion)
* O(n·lg(n)) time
* Not really adaptive

## *Discussion*

Heap sort is simple to implement, performs an O(n·lg(n)) in-place sort, but is not stable.

The first loop, the Θ(n) "heapify" phase, puts the array into heap order. The second loop, the O(n·lg(n)) "sortdown" phase, repeatedly extracts the maximum and restores heap order.

The sink function is written recursively for clarity. Thus, as shown, the code requires Θ(lg(n)) space for the recursive call stack. However, the tail recursion in sink() is easily converted to iteration, which yields the O(1) space bound.

Both phases are slightly adaptive, though not in any particularly useful manner. In the nearly sorted case, the heapify phase destroys the original order. In the reversed case, the heapify phase is as fast as possible since the array starts in heap order, but then the sortdown phase is typical. In the few unique keys case, there is some speedup but not as much as in shell sort or 3-way quicksort.

***Quick Sort***

It is a very popular sorting algorithm invented by C.A. R. Hoare in 1962. The name

comes from the fact that, in general, quick sort can sort a list of data elements

significantly faster than any of the common sorting algorithms. This algorithm is based

on the fact that it is faster and easier to sort two small arrays than one larger one. Quick

sort is based on divide and conquer method. quick sort is often the best practical choice

for sorting because it is efficient on average i.e. its expected running time is O(n log n)

and the constant factors hidden in O(n log n) are small. Its also having the advantage of

sorting in place and it works well in virtual memory environments. The most direct

competitor of quick sort is heap sort. Heap sort is typically somewhat slower than quick

sort, but the worst-case running time is always O (n log n). Quick sort also competes with

merge sort, another recursive sort algorithm but with the benefit of worst-case O (n log n)

running time. Merge sort is a stable sort, unlike quick sort and heap sort. The main

drawback of quick sort is that it achieves its worst case time complexity on data sets that

are common in practice (sequences that are already sorted or mostly sorted).To avoid

this, we modify quick sort so that it selects the pivot as a random element of the

sequence.

Quick sort is also known as partition-exchange sort. One of the elements is selected as the

partition element known as pivot element. The remaining items are compared to it and a

series of exchanges is performed. When the series of exchanges is done, the original

sequence has been partitioned into three sub sequences.

1. all items less than the pivot element

2. the pivot element in its final place

3. all items greater than the pivot element

At this stage, step 2 is completed and quick sort will be applied recursively to steps 1 and

3. The sequence is sorted when the recursion terminates.

13

Example of quick sort

a)

3 2 1 5 8 4 3 7

b)

1 2 3 5 8 4 3 7

c)

1 2 3 3 4 5 8 7

d)

1 2 3 3 4 5 7 8

e)

1 2 3 3 4 5 7 8

Fig 2.4 Example of Quick Sort

Algorithm

Quick sort (A, p, r)

1. If p < r

2. Then q partition (A, p ,r)

3. Quick sort ( A, p, q-1)

4. Quick sort (A ,q+1 ,r)

***Timsort:***

*Timsort* finds runs in the data, creates runs with insertion sort if necessary, and then uses merge sort to create the final sorted list. It has the same complexity (O(nlogn)) in the average and worst cases, but with pre-sorted data it goes down to O(n).

|  |  |
| --- | --- |
| Class | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| Data structure | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [Worst case performance](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n\log n)[[1]](http://en.wikipedia.org/wiki/Timsort#cite_note-1) |
| [Best case performance](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n) |
| [Average case performance](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n\log n) |
| [Worst case space complexity](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n) |

***Counting Sort***

This sorting algorithm sort the n input in the range 0 to k. The basic idea behind this

sorting algorithm is that find out the number of elements less than for each input elements

i.e. I, then only we find the final position of I. counting sort is use for integers in small

range.

This sorting algorithm must handle the situation in which several elements have the same

value i.e. duplicate value. An important property of counting sort is that it is stable i.e.

multiple keys with the same value are placed in the sorted array in the same order that

they appear in the input array. Counting sort beats the lower bound of O (nlogn) because

it is not a comparison sort. To sort the element using counting sort, we have to use three

arrays. One is for input array A [1...n], output array B [1…n] and other is C [0...k]

provides temporary working storage.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  |  |  |  | 3 |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 6 | 7 | 8 |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 0 | 2 | 3 | 0 | 1 |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 |  |  |  | 3 | 3 |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 2 | 4 | 7 | 7 | 8 |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | 3 |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 2 | 4 | 6 | 7 | 8 |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 5 | 7 | 8 |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 |

***Algorithm***

1. for I=0 to K

2. C[I]=0

3. For j=1 to length [A]

4. C [A(J)]=C[A(J)]+1

5. For I=1 to K

6. C[I]=C[I]+C[I-1]

7. For J=length(A) down to 1

8. B[C(A(J)]=A[J]

9. C[A(J)]=C[A(J)]-1

***Cocktail Sort :***

Cocktail sort, also known as bidirectional bubble sort, cocktail shaker sort, shaker sort (which can also refer to a variant of [selection sort](http://en.wikipedia.org/wiki/Selection_sort)), ripple sort, shuffle sort,[[1]](http://en.wikipedia.org/wiki/Cocktail_sort#cite_note-Duhl1986-1) shuttle sort or happy hour sort, is a variation of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) that is both a [stable](http://en.wikipedia.org/wiki/Stable_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) and a[comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). The algorithm differs from a [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) in that it sorts in both directions on each pass through the list. This sorting algorithm is only marginally more difficult to implement than a bubble sort, and solves the problem of [turtles](http://en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles) in bubble sorts

|  |
| --- |
| Cocktail sort |
|  |
| Class | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| Data structure | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [Worst case performance](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [Best case performance](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n) |
| [Average case performance](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(n^2) |
| [Worst case space complexity](http://en.wikipedia.org/wiki/Best%2C_worst_and_average_case) | O(1) |

procedure cocktailSort( A : list of sortable items ) defined as:

 do

 swapped := false

 for each i in 0 to length( A ) - 2 do:

 if A[ i ] > A[ i + 1 ] then // test whether the two elements are in the wrong order

 swap( A[ i ], A[ i + 1 ] ) // let the two elements change places

 swapped := true

 end if

 end for

 if swapped = false then

 // we can exit the outer loop here if no swaps occurred.

 break do-while loop

 end if

 swapped := false

 for each i in length( A ) - 2 to 0 do:

 if A[ i ] > A[ i + 1 ] then

 swap( A[ i ], A[ i + 1 ] )

 swapped := true

 end if

 end for

 while swapped // if no elements have been swapped, then the list is sorted

end procedure

# *Comb sort*

Comb sort is a [comparison](http://code.wikia.com/wiki/Comparison_sort?action=edit&redlink=1) [sorting algorithm](http://code.wikia.com/wiki/Sorting_algorithm?action=edit&redlink=1). It is an exchange sort, similar to [bubble sort](http://code.wikia.com/wiki/Bubble_sort).

In comb sort, *gaps* (distance of two items from each other) are introduced. The gap in bubble sort is 1. The gap starts out as a large value, and, after each traversal, the gap is lessened, until it becomes 1, where the algorithm basically degrades to a bubble sort. This idea can practically *kill* turtles because some of them would "jump" to the beginning of the list early on.

The *shrink factor* determines how much the gap is lessened. This value is crucial because a small value means that it would be slower for the gap to degrade to 1, slowing down the process, while a large value will not effectively kill turtles. An ideal shrink factor is 1.3.

## *Algorithm*

comb\_sort(list of t)

 gap = list.count

 temp as t

 swapped = false

 while gap > 1 or not swapped

 swapped = false

 if gap > 1 then

 gap = floor(gap/1.3)

 i = 0

 while i + gap < list.count

 if list(i) > list(i + gap)

 temp = list(i) // swap

 list(i) = list(i + gap)

 list(i + gap) = temp

 i += 1

Conclusion:

As we have seen there are many different sorting algorithm in data structure and we identified in this report on the twelve algorithm sort, which it has different

References:

<http://careerride.com/Data-structure-bubble-quick-sort.aspx>

 <http://wiki.answers.com/Q/What_are_advantages_and_disadvantages_of_bubble_sort>

<http://www.techopedia.com/definition/3757/bubble-sort>

<http://freepdfdb.com/pdf/the-bubble-sort-61808711.html>

<http://www.sorting-algorithms.com/quick-sort>

<http://en.wikipedia.org/wiki/Insertion_sort#Algorithm>

<http://www.ehow.com/info_8446142_advantages-disadvantages-sorting-algorithms.html>

<http://www.cprogramming.com/tutorial/computersciencetheory/radix.html>

<https://en.wikipedia.org/wiki/Sorting_algorithm>

<http://en.wikipedia.org/wiki/Timsort>

<http://en.wikipedia.org/wiki/Cocktail_sort>