🙞 **SELECTION SORT** 🙜

🞛 Algorithm Clarification ([[1]](#footnote-1)):

A comparison sort based on comparing and swapping of two elements in each pass. Firstly; we select the largest (or smallest depending on sorting order) value in the array as in figure[1] and swap with the last element of the array, in case that the last element is not the largest value. Now, we have two parts in the array, sorted one includes the largest value and unsorted one includes other smaller values in the array which we will work on to sort the array. Assume we shrink the array size by one; we will obtain the unsorted part! Now select the largest value in the unsorted part and swap with the last element of this part, as in first swap. Till this moment, we have two sorted elements in the sorted part. By processing this procedure on the remaining elements in the unsorted part; we will sort the whole array.



🞛 JAVA Code ([[2]](#footnote-2)):

public static void selectionSort(int[] x) {

for (int i=x.length-1; i>=0; i--) {

int maxIndex = i; // Index of largest remaining value.

for (int j=0; j<i; j++) {

if (x[maxIndex] < x[j]) {

maxIndex = j; // Remember index of new maximum

}

}

if (maxIndex != i) {

//... Exchange current element with largest remaining. int temp = x[i];

x[i] = x[maxIndex];

x[maxIndex] = temp;

}

}

}

🞛 Time Analysis[[3]](#footnote-3):

 

🙞 **GNOME SORT** 🙜

🞛 Algorithm Clarification:

Gnome sort (Stupid sort), originally proposed by *Dr.* *Hamid Sarbazi-Azad* (Professor of Computer Engineering at *Sharif University of Technology*) in 2000 and called Stupid sort (not to be confused with Bogosort), then described by Dick Grune and named “Gnome sort”**([[4]](#footnote-4))**. This form does NOT require nested loops, but requires a series of swaps to accomplish movement of an element to its proper place (as in Bubble Sort).

First of all, we need a variable such as ‘pos’ to hold the value of current position in the array (starts from index=1). Then we compare element of position [pos] and element of position [pos-1], we swap them (if needed) and compare ‘pos’ with the value ‘1’ to decide whether increment pos (pos<=1) or decrement it (pos>1). By repeating this procedure we will sort the elements of the array. Table[1] is an illustrative example for this algorithm:

🞛 JAVA Code ([[5]](#footnote-5)):

public static int[] gnomeSort(int[] nums){ //takes unsorted array, returns sorted

 int index=1; //start of search

 int temp;

 while(index<nums.length){ //until the array is fully sorted

 if(nums[index]<nums[index-1]){ //compares nums[index] with nums[index-1]. if smaller, switch.

 temp=nums[index];

 nums[index]=nums[index-1];

 nums[index-1]=temp;

 index--; //must decrease index to recheck. since they have been swapped, the array may still be out of order

 if(index==0){ //prevents index from going lower than 1

 index=1;

 }

 }

 else{

 index++; //if sorted, go up

 }

 }

 return(nums); //reaching the end of the array- completely sorted, returns nums

 }

🞛 Time Analysis ([[6]](#footnote-6)):

As a difference from Insertion sort; in Gnome sort, moving an element to its proper place is accomplished by a series of swaps, as in bubble sort. The name comes from the supposed behavior of the Dutch garden gnome in sorting a line of flowerpots. Although it has no nested loops; The running time is O(n2), and in practice the algorithm has been reported to run slightly slower than bubble sort, although this depends on the details of the code and its implementation.

🙞 **STRAND SORT** 🙜

🞛 Algorithm Clarification ([[7]](#footnote-7)):

Assume we have the following linked list and we need to sort its data using *Strand Sort*;



1. Strand sort is another form of comparison sorts, its procedure consists of these steps:
2. Parse the unsorted list once, taking out any ascending (sorted) numbers.
3. The (sorted) sublist is, for the first iteration, pushed onto the empty sorted list.
4. Parse the unsorted list again, again taking out relatively sorted numbers.
5. Since the sorted list is now populated, merge the sublist into the sorted list.
6. Repeat steps 3–4 until both the unsorted list and sublist are empty.

🞛 JAVA Code ([[8]](#footnote-8)):

import java.util.Arrays;

import java.util.LinkedList;

 public class Strand{

 // note: the input list is destroyed

 public static <E extends Comparable<? super E>>

 LinkedList<E> strandSort(LinkedList<E> list){

 if(list.size() <= 1) return list;

 LinkedList<E> result = new LinkedList<E>();

 while(list.size() > 0){

 LinkedList<E> sorted = new LinkedList<E>();

 sorted.add(list.removeFirst()); //same as remove() or remove(0)

 for(Iterator<E> it = list.iterator(); it.hasNext(); ){

E elem = it.next();

 if(sorted.peekLast().compareTo(elem) <= 0){

 sorted.addLast(elem); //same as add(elem) or add(0, elem)

 it.remove();

 }

 }

 result = merge(sorted, result);

 }

 return result;

 }

 private static <E extends Comparable<? super E>>

 LinkedList<E> merge(LinkedList<E> left, LinkedList<E> right){

 LinkedList<E> result = new LinkedList<E>();

 while(!left.isEmpty() && !right.isEmpty()){

 //change the direction of this comparison to change the direction of the sort

 if(left.peek().compareTo(right.peek()) <= 0)

 result.add(left.remove());

 else

 result.add(right.remove());

 }

 result.addAll(left);

 result.addAll(right);

 return result;

 }

}

🞛 Time Analysis ([[9]](#footnote-9)):

The strand sort algorithm is O(n2) in the average case. In the best case (a list which is already sorted) the algorithm is linear, or O(n). In the worst case (a list which is sorted in reverse order) the algorithm is O(n2).

Strand sort is more powerful for linked lists than arrays, since this sort requires frequent insertions and removals of data which take constant time in linked lists and O(n) in arrays. Strand sort is also useful for data which already has large amounts of sorted data, because such data can be removed in a single strand.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Worst case  | O(n2) | Best Case | O(n) | Average case | O(n2) |

🙞 **COMB SORT** 🙜

🞛 Algorithm Clarification ([[10]](#footnote-10)):

A relatively simple sorting algorithm originally designed by Włodzimierz Dobosiewicz in 1980. Later, Stephen Lacey and Richard Box rediscovered it in 1991.

Comb sort improves on bubble sort. In other words, it eliminates smaller values near the end of the unsorted array. In bubble sort, the distance (gap) between any two elements is to be compared is always 1; but in comb sort the gap can be much more than 1. Here, the gap starts out as the length of the array being sorted divided by a factor called shrink factor (experimentally determined to be 1.3), and then we sort the array with that value as the first stage of sorting (rounded down to an integer if needed). Then the gap itself divided by the shrink factor again, and we sort the array using this new gap as second stage. By repeating this procedure we will finally reach a case in which the gap is 1. At this point, comb sort continues and accomplish the last stage using a gap of 1 (bubble sort) until the list is fully sorted.

🞛 JAVA Code ([[11]](#footnote-11)):

public static <E extends Comparable<? super E>> void sort(E[] input) {

 int gap = input.length;

 boolean swapped = true;

 while (gap > 1 || swapped) {

 if (gap > 1) {

 gap = (int) (gap / 1.3);

 }

 swapped = false;

 for (int i = 0; i + gap < input.length; i++) {

 if (input[i].compareTo(input[i + gap]) > 0) {

 E t = input[i];

 input[i] = input[i + gap];

 input[i + gap] = t;

 swapped = true;

 }

 }

 }

}

🞛 Time Analysis ([[12]](#footnote-12)):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Worst case  | \Omega(n^2) | Best Case | O(n) | Average case | \Omega(n^2/2^p),*p:* # of increments |

🙞 **BINARY SEARCH TREE SORT** 🙜

🞛 Algorithm Clarification ([[13]](#footnote-13)):

This type constructs a binary search tree (left < root <= right) or (right < root <= left). In other words, we first insert the root and then when we insert other elements we need to compare to the root and any other elements in the tree. So, we need at maximum (n-1) comparisons to insert an element where n is the number of data in binary search tree. But the problem is; if we want to print or even read the tree as sorted we need to use inorder procedure which has also extra cost of time.

🞛 JAVA Code ([[14]](#footnote-14)):

public NodeT insert(NodeT nt, NodeT rt)

 {

 if(rt == null)

 {

 rt = nt;

 rt.left = rt.right = null;

 }

 else{

 if(nt.getEnglish().compareToIgnoreCase(rt.getEnglish()) >= 0)

 rt.right = insert(nt, rt.right);

 else

 rt.left = insert(nt, rt.left);

 }

 return rt;

public ArrayList<String> print(NodeT nt)

 {

 if(nt!=null)

 {

 print(nt.left);

 printList.add(nt.getEnglish() + nt.getArabic());

 print(nt.right);

 }

 return printList;

 }

🞛 Time Analysis ([[15]](#footnote-15)):

In first step we need constant time to insert the root since no comparisons are to be done. Now, inserting new elements needs at maximum (n-1) comparisons to insert an element where n is the number of data in binary search tree, T(n) (comparison) = 1+2+…+(n-1).

T(n) = Sum(i), i = 1+2+…+(n-1) = n(n-1)/2 🡪 T(n) = O(n2). Not that whatever the situation of data, comparisons are to be done in each stage, which mean worst, best and Average cases have same time.

🙞 **REFERENCES** 🙜

**(\*) Complexity of Insertion Sort, PowerPoint Slides, [Accessed: Jan29, 2014] <**[**http://www.csie.ntu.edu.tw/~sylee/courses/ds/lec2.ppt**](http://www.csie.ntu.edu.tw/~sylee/courses/ds/lec2.ppt)**>**

**(\*) Selection Sort, More efficient selection sort - Move every value only once (edited), [Accessed: Jan29, 2014] <**[**http://www.leepoint.net/notes-java/data/arrays/31arrayselectionsort.html**](http://www.leepoint.net/notes-java/data/arrays/31arrayselectionsort.html) **>**

**(\*) InsertionSortBubbleSortSelectionSort, [Accessed: Jan29, 2014] <**[**http://www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt**](http://www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt)**>**

**(\*) Gnome sort, Wikipedia: the free encyclopedia, [Accessed: Jan31, 2014] <**[**http://en.wikipedia.org/wiki/Gnome\_sort**](http://en.wikipedia.org/wiki/Gnome_sort) **>**

**(\*) Gnome Sort, Professor Java, [Accessed: Jan31, 2014] <**[**http://professorjava.weebly.com/gnome-sort.html**](http://professorjava.weebly.com/gnome-sort.html) **>**

**(\*) The Gnome Sort, Natural Blogarithms, [Accessed: Jan31, 2014] <**[**http://webcache.googleusercontent.com/search?q=cache:http://blog.drscottfranklin.net/2006/01/12/the-gnome-sort/**](http://webcache.googleusercontent.com/search?q=cache:http://blog.drscottfranklin.net/2006/01/12/the-gnome-sort/) **>**

**(\*) Strand Sort, Wikipedia: the free encyclopedia, [Accessed: Jan31, 2014] <**[**http://en.wikipedia.org/wiki/Strand\_sort**](http://en.wikipedia.org/wiki/Strand_sort) **>**

**(\*) Sorting algorithms/Strand sort, [Accessed: Jan31, 2014] <**[**http://rosettacode.org/wiki/Sorting\_algorithms/Strand\_sort**](http://rosettacode.org/wiki/Sorting_algorithms/Strand_sort) **>**

 **(\*) Comb Sort, Wikipedia: the free encyclopedia, [Accessed: Feb2, 2014] <**[**http://en.wikipedia.org/wiki/Comb\_sort**](http://en.wikipedia.org/wiki/Comb_sort) **>**

**(\*) Sorting-algorithms-Comb-sort, GitHub, RosettaCodeData, [Accessed: Feb2, 2014] <**[**https://github.com/acmeism/RosettaCodeData/blob/master/Task/Sorting-algorithms-Comb-sort/Java/sorting-algorithms-comb-sort.java**](https://github.com/acmeism/RosettaCodeData/blob/master/Task/Sorting-algorithms-Comb-sort/Java/sorting-algorithms-comb-sort.java)**>**

**(\*) Data Structure and Algorithm, Class notes.**

1. **() Complexity of Insertion Sort, PowerPoint Slides, [Accessed: Jan29, 2014] <**[**http://www.csie.ntu.edu.tw/~sylee/courses/ds/lec2.ppt**](http://www.csie.ntu.edu.tw/~sylee/courses/ds/lec2.ppt)**>** [↑](#footnote-ref-1)
2. **() Selection Sort, More efficient selection sort - Move every value only once (edited), [Accessed: Jan29, 2014] <**[**http://www.leepoint.net/notes-java/data/arrays/31arrayselectionsort.html**](http://www.leepoint.net/notes-java/data/arrays/31arrayselectionsort.html) **>** [↑](#footnote-ref-2)
3. **()InsertionSortBubbleSortSelectionSort, [Accessed: Jan29, 2014] <**[**http://www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt**](http://www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt)**>** [↑](#footnote-ref-3)
4. **() Gnome sort, Wikipedia: the free encyclopedia, [Accessed: Jan31, 2014] <**[**http://en.wikipedia.org/wiki/Gnome\_sort**](http://en.wikipedia.org/wiki/Gnome_sort) **>** [↑](#footnote-ref-4)
5. **() Gnome Sort, Professor Java, [Accessed: Jan31, 2014] <**[**http://professorjava.weebly.com/gnome-sort.html**](http://professorjava.weebly.com/gnome-sort.html) **>** [↑](#footnote-ref-5)
6. **() Natural Blogarithms, The Gnome Sort, [Accessed: Jan31, 2014] <**[**http://webcache.googleusercontent.com/search?q=cache:http://blog.drscottfranklin.net/2006/01/12/the-gnome-sort/**](http://webcache.googleusercontent.com/search?q=cache:http://blog.drscottfranklin.net/2006/01/12/the-gnome-sort/)> [↑](#footnote-ref-6)
7. **()Strand Sort, Wikipedia: the free encyclopedia, [Accessed: Jan31, 2014] <**[**http://en.wikipedia.org/wiki/Strand\_sort**](http://en.wikipedia.org/wiki/Strand_sort) **>** [↑](#footnote-ref-7)
8. **() Sorting algorithms/Strand sort, [Accessed: Jan31, 2014] <**[**http://rosettacode.org/wiki/Sorting\_algorithms/Strand\_sort**](http://rosettacode.org/wiki/Sorting_algorithms/Strand_sort) **>** [↑](#footnote-ref-8)
9. **() Strand Sort, Wikipedia: the free encyclopedia, [Accessed: Jan31, 2014] <**[**http://en.wikipedia.org/wiki/Strand\_sort**](http://en.wikipedia.org/wiki/Strand_sort) **>** [↑](#footnote-ref-9)
10. **() Comb Sort, Wikipedia: the free encyclopedia, [Accessed: Feb2, 2014] <**[**http://en.wikipedia.org/wiki/Comb\_sort**](http://en.wikipedia.org/wiki/Comb_sort) **>** [↑](#footnote-ref-10)
11. **() Sorting-algorithms-Comb-sort, GitHub, RosettaCodeData, [Accessed: Feb2, 2014] <**[**https://github.com/acmeism/RosettaCodeData/blob/master/Task/Sorting-algorithms-Comb-sort/Java/sorting-algorithms-comb-sort.java**](https://github.com/acmeism/RosettaCodeData/blob/master/Task/Sorting-algorithms-Comb-sort/Java/sorting-algorithms-comb-sort.java)**>** [↑](#footnote-ref-11)
12. **() Comb Sort, Wikipedia: the free encyclopedia, [Accessed: Feb2, 2014] <**[**http://en.wikipedia.org/wiki/Comb\_sort**](http://en.wikipedia.org/wiki/Comb_sort) **>** [↑](#footnote-ref-12)
13. **() Data Structure and Algorithm, Class notes.** [↑](#footnote-ref-13)
14. **() Data Structure and Algorithm, Class notes.** [↑](#footnote-ref-14)
15. **() Data Structure and Algorithm, Class notes.** [↑](#footnote-ref-15)