BIRZEIT UNIVERSITY

Department of Computer Science

COMP242

Data Structure and Algorithm

Project #4

theoretical part
a report about 5 new sorting algorithms

Instructor : Iyad Jaber

Section: #1

Name : Maryam Shaheen

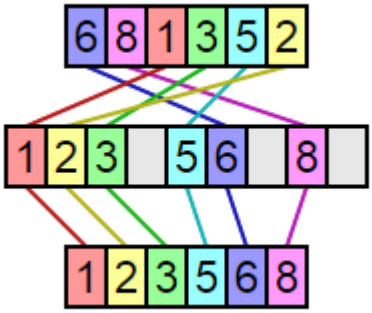
ID# : 1140427

Contents

5 Sorting Algorithms

1.counting sort:

Counting sort works by creating an auxiliary array the size of the range of values, the unsorted values are then placed into the new array using the *value* as the *index*. The auxiliary array is now in sorted order and can be iterated over to construct the sorted array.



Counting sort can be exceptionally fast because of the way that elements are sorted using their values as array keys. This means that more memory is required for the extra array at the cost of running time.
It runs in **O(n + k)** ***O(n+k)*** time where ***n*** is the number of elements to be sorted and ***k*** is the number of possible values in the range.

Complexity :

Time			*Space*
Worst case	***Best case***	***Average case***	***Worst case***
O(n+k)	*O(n + k)O(n+k)*	*O(n + k)O(n+k)*	*O(k) auxiliary*

Analysis:

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most ***k + 1*** times and therefore take ***O(k)*** time. The other two for loops, and the initialization of the output array, each take ***O(n)*** time. Therefore, the time for the whole algorithm is the sum of the times for these steps, ***O(n + k).***

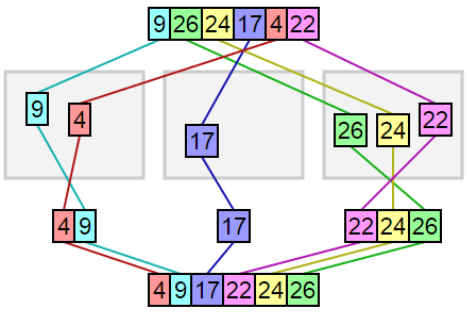
Because it uses arrays of length ***k + 1*** and n, the total space usage of the algorithm is also ***O(n + k).*** For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the Count array which uses space ***O(k).***

Algorithm :

```
# variables:
#    input -- the array of items to be sorted; key(x) returns the key for item x
#    n -- the length of the input
#    k -- a number such that all keys are in the range 0..k-1
#    count -- an array of numbers, with indexes 0..k-1, initially all zero
#    output -- an array of items, with indexes 0..n-1
#    x -- an individual input item, used within the algorithm
#    total, oldCount, i -- numbers used within the algorithm

# calculate the histogram of key frequencies:
for x in input:
    count[key(x)] += 1

# calculate the starting index for each key:
total = 0
for i in range(k):    # i = 0, 1, ... k-1
    oldCount = count[i]
    count[i] = total
    total += oldCount

# copy to output array, preserving order of inputs with equal keys:
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1

return output
```

2.Bucket Sort :

Bucket sort can be exceptionally fast because of the way elements are assigned to buckets, typically using an array where the index is the value. This means that more auxiliary memory is required for the buckets at the cost of running time than more comparison sorts. It runs in ***O(n+k)O(n+k)*** time in the average case where ***n*** is the number of elements to be sorted and ***k*** is the number of buckets.



Complexity:

	Time		*Space*
Worst case	***Best case***	***Worst case***	***Best case***
$O(n^2)$	***O(n + k)***	***O(n+k)***	***O(n + k)***

Analysis:

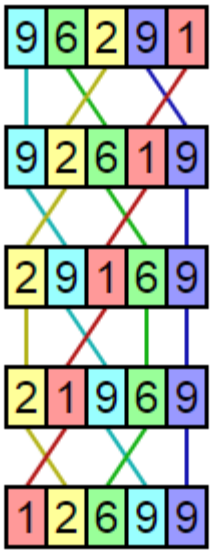
- In this sorting algorithm we create buckets and put elements into them
- Then we apply some sorting algorithm (insertion sort) to sort the element in each bucket
- Finally, we take the elements out and join them to get the sorted result.

Algorithm:

```
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]
```

Here *array* is the array to be sorted and *n* is the number of buckets to use. The function *msbits(x,k)* returns the *k* most significant bits of *x* (*floor(x/2^(size(x)-k))*); different functions can be used to translate the range of elements in *array* to *n* buckets, such as translating the letters A–Z to 0–25 or returning the first character (0–255) for sorting strings. The function *nextSort* is a sorting function; using *bucketSort* itself as *nextSort* produces a relative of radix sort; in particular, the case *n = 2* corresponds to quicksort(although potentially with poor pivot choices).

3. Odd-Even Sort:



Like **bubble sort**, odd-even sort works by iterating through the list, comparing adjacent elements and swapping them if they're in the wrong order. The unique characteristic of odd-even sort, and also how it got its name, is how the sort's iterations alternate between sorting odd/even and even/odd indexed pairs.

Much like bubble sort, odd-even sort has very little relevance in the real world and is mainly used to teach algorithms.

Complexity:

Time			*Space*
Worst case	***Best case***	***Worst case***	***Best case***
$O(n^2)$	$\Theta(n)$	$O(n^2)$	$O(1)$

Example:

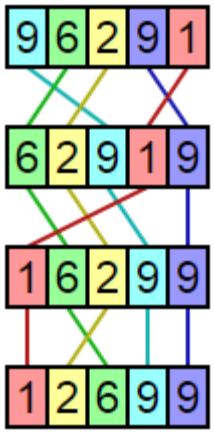
You can see from the following example how ridiculous the amount of passes needed to sort the array.

Array	Passes
3 – 99 – 25 – 4 – 1 – 2 – 46 – 32 – 23	Initial Array
3 – 25 – 99 – 1 – 4 – 2 – 46 – 23 - 32	Odd/Even Indexed Pairs Compared
3 – 25 – 1 – 99 – 2 – 4 – 23 – 46 – 32	Even/Odd Indexed Pairs Compared
3 – 1 – 25 – 2 – 99 – 4 – 23 – 32 – 46	Odd/Even Indexed Pairs Compared
1 – 3 – 2 – 25 – 4 – 99 – 23 – 32 – 46	Even/Odd Indexed Pairs Compared
1 – 2 – 3 – 4 – 25 – 23 – 99 – 32 – 46	Odd/Even Indexed Pairs Compared
1 – 2 – 3 – 4 – 23 – 25 – 32 – 99 – 46	Even/Odd Indexed Pairs Compared
1 – 2 – 3 – 4 – 23 – 25 – 32 – 46 – 99	Odd/Even Indexed Pairs Compared
1 – 2 – 3 – 4 – 23 – 25 – 32 – 46 – 99	Even/Odd Indexed Pairs Compared sorted = true, stop the loop

Algorithm :

```
function oddEvenSort(list) {
  function swap( list, i, j ){
    var temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  }

  var sorted = false;
  while(!sorted)
  {
    sorted = true;
    for(var i = 1; i < list.length-1; i += 2)
    {
      if(list[i] > list[i+1])
      {
        swap(list, i, i+1);
        sorted = false;
      }
    }

    for(var i = 0; i < list.length-1; i += 2)
    {
      if(list[i] > list[i+1])
      {
        swap(list, i, i+1);
        sorted = false;
      }
    }
  }
```

4. Cocktail Sort:



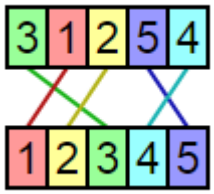
Like bubble sort, cocktail sort works by iterating through the list, comparing adjacent elements and swapping them if they're in the wrong order. The only real difference is that it alternates directions instead of only going from left to right. Because of this, cocktail sort manages to get around the "turtles problem" of bubble sort, however it still retains the same worst case computational complexity.

Much like bubble sort, cocktail sort has very little relevance in the real world and is mainly used to teach algorithms.

Complexity:

	Time		***Space***
Worst case	***Best case***	***Worst case***	***Best case***
$O(n^2)$	***O (n)***	***$O(n^2)$***	***O(1)***

When it's fast:



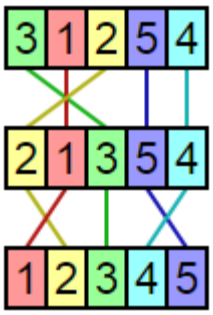
Cocktail sort is at its fastest when it can reach a sorted list with a minimal number of passes. Since only adjacent elements are swapped, this means that cocktail sort performs best when elements are physically nearby their sorted positions.

Algorithm:

```
public static void cocktailSort( int[] A ){
        boolean swapped;
        do {
                swapped = false;
                for (int i =0; i<=  A.length  - 2;i++) {
                        if (A[ i ] > A[ i + 1 ]) {
                                //test whether the two elements are in the wrong order
                                int temp = A[i];
                                A[i] = A[i+1];
                                A[i+1]=temp;
                                swapped = true;
                        }
                }
                if (!swapped) {
                        //we can exit the outer loop here if no swaps occurred.
                        break;
                }
                swapped = false;
                for (int i= A.length - 2;i>=0;i--) {
                        if (A[ i ] > A[ i + 1 ]) {
                                int temp = A[i];
                                A[i] = A[i+1];
                                A[i+1]=temp;
                                swapped = true;
                        }
                }
                //if no elements have been swapped, then the list is sorted
        } while (swapped);
}
```

5. comb Sort:

Comb sort is similar to bubble sort in that is iterates through the list multiple times, swapping elements that are out of order as it goes. The difference is that comb sort doesn't start off looking at adjacent elements but instead looks at elements a certain number of indexes apart, this is called the gap. The gap is defined as $\lfloor n/c \rfloor$, where n is the number of elements and cc is the shrink factor which is typically set as 1.3. After each iteration, this number is against divided by cc and floored until eventually the algorithm is looking at adjacent elements.



Similar to cocktail sort, comb sort improves upon bubble sort due to its ability to deal with the "turtles problem". It does this by swapping elements that are separated by many indexes by a while the gap is large and progressively shifts them closer to their correct index as the algorithm continues.

Complexity:

Time		*Space*
Worst case	*Best case*	*Worst case*
$O(n^2)$	$\Theta(nlogn)$*	*O(1)*

Algorithm:

```
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;
            }
        }
    }
}
```

Analysis:

As can be seen from the code, the comb sort sorts the array with a relatively big gap, then shrinks the gap (by dividing by 1.3) and repeat the same process, until it reaches gap equal to 1, which would transform into bubble sort at this point.

It should be mentioned that the shrink factor has a great effect on comb sort, and 1.3 was chosen as the best factor experimentally after testing on comb sort over 200000 random lists. (unstable)

Summary:

Name	*BestCase*	*AverageCase*	*WorstCase*	*Space Complexity*	*Stable*	*Type*
Count Sort	***O(n + k)O(n+k)***	***O(n + k)O(n+k)***	***O(n+k)***	***O(k)***	***No***	***-***
Bucket Sort	***O(n+k)***	***O(n+k)***	***$O(n^2)$***	***O(n.k)***	***-***	***-***
Odd-Even Sort	***O(n)***	***$O(n^2)$***	***$O(n^2)$***	***O(1)***	***Yes***	***Comparison***
Cocktail Sort	***O(n)***	***$O(n^2)$***	***$O(n^2)$***	***O(1)***	***Yes***	***Comparison***
Comb Sort	***O(n)***	***O(nlogn)***	***$O(n^2)$***	***O(1)***	***No***	***Comparison***

References:

all information's are brought from:
http://www.growingwiththeweb.com
https://www.wikipedia.org