**

**Faculty Of Engineering And Technology**

**Computer Science Department**

**COMP242**

**Data Structures**

**Project No. 4**

**5 Sorting Algorithms**

**Student Name: Mohammad Ibrahim**

**Student Number: 1141618**

**Instructor Name: Dr. Mamoun Nawahdah**

**Date: 20-5-2016**

**Section: 3**

Contents

**Algorithm 1:** **Comb Sort ……….………………………………………………………... 3**

**Algorithm 2: Gnome Sort ………………………………………………………………… 4**

**Algorithm 3: Bucket Sort…..………………………………….………………………… 5**

**Algorithm 4: Count Sort……….…………………………………………………………. 6**

**Algorithm 5: Odd-Even Sort………………………………………………………….… 8**

**Summary ……………………………………………………………………………………….. 9**

**References …………………………………………………………………………………... 10**

# **Algorithm 1: Comb Sort[1]**

* **Definition**

A simple sorting algorithm. The basic idea is simple. It’s to get rid of small values near the end of the list (*turtles*) as they usually cause trouble for bubble sort, unlike *rabbits*, large values at the beginning of the list, which do not cause much trouble.

In bubble sort, the gap is always 1, but in comb sort, the gap can be bigger than 1. This indicates that it is a modified version of bubble sort.

* **Algorithm/Pseudocode**

function combSort(a[]) {

gap = length(a);

shrink = 1.3;

while (gap > 1 OR swapped == true) {

gap = (int) gap / shrink; // INTEGER DIVISION!

if (gap < 1)

gap = 1;

swapped = false;

for (i = 0 , i + gap < length(a) , i = i + 1) {

if (a[i] > a[i + gap]) {

exchange(a[i], a[i + gap]);

swapped = true;

} else

if (a[i] == a[i + gap])

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. The time analysis is **O(n)** in best case, **O(n logn)** in average cases and **O(n2)** in worst cases. As for worst case space complexity, **O(1)**

# **Algorithm 2: Gnome Sort[2]**

* **Definition**

Also known as **bin sort**, is a sorting algorithm that 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.

* **Algorithm/Pseudocode**

function gnomeSort(a[]) {

pos = 0;

while (pos < length(a)) {

if ((pos == 0) OR (a[pos – 1] <= a[pos]))

pos = pos + 1;

else {

exchange(a[pos], a[pos – 1]);

pos = pos – 1;

}

}

}

* **Analysis**

As it is seen, the algorithm is quite simple, and no nested loops are needed.

For time complexity, the worst case of the algorithm is **O(n2)**. Here is the proof:

Suppose n is the number of elements in the array a, so it iterates n times in the while loop, however, if 2 elements are found in wrong order, it goes back one position, and since we are talking about worst case, it would return until it reaches the first index every time, thus iterating back n times in the worst case.

The average case is similar to the worst case **O(n2)**, while the best case is **O(n)**, making it very similar to insertion sort. The time complexity in worst case is **O(1)**.

It’s obvious that the algorithm isn’t a good choice for a large number of elements. However, it is efficient for low number of elements, as it is fast and tends toward **O(n)** when the array is initially almost sorted. Also, the code of the algorithm is tiny and simple.

It appears that is a lot similar to insertion sort in behavior. Even some people say it is a variant of insertion sort, but with its logic hidden in a single loop.

# **Algorithm 3: Bucket Sort[3]**

* **Definition**

Also known as **Stupid Sort**, and sometimes called “the simplest sorting algorithm”, is a stable sorting algorithm similar to insertion sort, but what’s different is that elements are swapped to move an element to its proper place. The algorithm always finds the first two adjacent elements that are in wrong order and swaps them. It only checks the position directly previous to the swapped elements.

* **Algorithm/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]

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](https://en.wikipedia.org/wiki/Radix_sort); in particular, the case *n = 2* corresponds to [quicksort](https://en.wikipedia.org/wiki/Quicksort)(although potentially with poor pivot choices).

* **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.

|  |  |
| --- | --- |
| [**Worst case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Best case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | \Omega(n+k) |
| [**Average case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | \Theta(n+k) |
| [**Worst case space complexity**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\cdot k) |

# **Algorithm 4: Count Sort[4]**

* **Definition**

 is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for [sorting](https://en.wikipedia.org/wiki/Sorting_algorithm) a collection of objects according to keys that are small [integers](https://en.wikipedia.org/wiki/Integer); that is, it is an [integer sorting](https://en.wikipedia.org/wiki/Integer_sorting) algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items

* **Algorithm/Pseudocode**
* *# 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
* **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 5: Odd-Even Sort[5]**

* **Definition**

Also known as **Brick Sort**, is yet another simple sorting algorithm. It is a comparison sort that is similar to bubble sort. Its basic idea is to compare between odd or even indexed elements and switch if they are wrong in order. Next, it alternates between even and odd indexes until the array is sorted.

* **Algorithm/Pseudocode**

function oddEvenSort(a[]) {

sorted = false;

while (!sorted) {

sorted = true;

for (i = 1 , i < length(a) – 1 , i = i + 2) {

if (a[i] > a[i + 1]) {

exchange(a[i], a[i + 1]);

sorted = false;

}

}

for (i = 0 , i < length(a) – 1 , i = i + 2) {

if(a[i] > a[i + 1]) {

exchange(a[i], a[i + 1]);

sorted = false;

}

}

}

}

* **Analysis**

Running time in worst case is **O(n2)**, best case is **O(n)** and average is **O(n2)**. Stable, and has **O(1)** space complexity for worst case.

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

**Summary[6]**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Name | Best-Case | Average-Case | Worst-Case | Space-Complexity | Stable | Type |
| Comb Sort | O(n) | O(nlogn) | O(n2) | O(1) | No | Comparison |
| Gnome Sort | O(n) | O(n2) | O(n2) | O(1) | Yes | Comparison |
| Bucket Sort | O(n+k) | O(n+k) | O(n2) | O(n.k) | - | - |
| Count Sort | - | - | - | - | - | - |
| Odd-Even Sort | O(n) | O(n2) | O(n2) | O(1) | Yes | Comparison |

**References**

1. Comb Sort, Wikipedia: the free encyclopedia, [Accessed: May 21, 2016]

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

1. Paul E. Black, "gnome sort", in [Dictionary of Algorithms and Data Structures](http://www.nist.gov/dads/), Vreda Pieterse and Paul E. Black, eds. 6 October 2010. [Accessed: May 21, 2016] <<http://www.nist.gov/dads/HTML/gnomeSort.html>>
2. Bucket Sort, Wikipedia: the free encyclopedia, [Accessed: May 21, 2016]

[< https://en.wikipedia.org/wiki/Bucket\_sort>](%20https:/en.wikipedia.org/wiki/Bucket_sort)

1. Counting Sort, Wikipedia: the free encyclopedia, [Accessed: May 21, 2016]

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

1. Odd-even sort, Wikipedia: the free encyclopedia, [Accessed: May 22, 2016], < <https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort>>
2. Summary, Wikipedia: the free encyclopedia, [Accessed: May 22, 2016], < <https://en.wikipedia.org/wiki/Sorting_algorithm>>