

Islam Jihad Joma

1191375

Section 3

* **Counting Sort**

*count* = array of k+1 zeros

**for** *x* **in** *input* **do**

 count[key(*x*)] += 1

*total* = 0

**for** *i* **in** 0, 1, ... k **do**

 count[i], *total* = *total*, count[*i*] + *total*

*output* = array of the same length as input

**for** *x* **in** *input* **do**

 *output*[*count*[key(*x*)]] = *x*

 *count*[key(*x*)] += 1

**return** *output*

\*1

It finds the maximum element from the given array and then initialize an array of length max+1 with all elements equal to 0. This array is used for storing the count of the elements in the array.

Then store the count of each element at their respective index in count array

For example: if the count of element 7 is 3 then, 3 is stored in the 7th position of count array. If element "9" is not present in the array, then 0 is stored in 9th position and so on.

The 4th step is to store cumulative sum of the elements of the index count array then find the index of each element of the original array in the count array. This gives the cumulative count. And place the element in the index calculated-1

After placing each element at its correct position, decrease its count by one

\*2

And here is the C code \*3

1. #include <stdio.h>
2.
3. */\* Counting sort function \*/*
4. void counting\_sort(int A[], int k, int n)
5. {
6. int i, j;
7. int B[15], C[100];
8. for (i = 0; i <= k; i++)
9. C[i] = 0;
10. for (j = 1; j <= n; j++)
11. C[A[j]] = C[A[j]] + 1;
12. for (i = 1; i <= k; i++)
13. C[i] = C[i] + C[i-1];
14. for (j = n; j >= 1; j--)
15. {
16. B[C[A[j]]] = A[j];
17. C[A[j]] = C[A[j]] - 1;
18. }
19. printf("The Sorted array is : ");
20. for (i = 1; i <= n; i++)
21. printf("%d ", B[i]);
22. }
23. */\* End of counting\_sort() \*/*
24.
25. */\* The main() begins \*/*
26. int main()
27. {
28. int n, k = 0, A[15], i;
29. printf("Enter the number of input : ");
30. scanf("%d", &n);
31. printf("**\n**Enter the elements to be sorted :**\n**");
32. for (i = 1; i <= n; i++)
33. {
34. scanf("%d", &A[i]);
35. if (A[i] > k) {
36. k = A[i];
37. }
38. }
39. counting\_sort(A, k, n);
40. printf("**\n**");
41. return 0;
42. }

time complexity: O(n+k) in best, worst and average cases. k is the range of the array

space complexity: O(k)

stability: yes it’s stable were k is the range the array

 we could not sort the elements if we have negative numbers in it. Because there are no negative array indices

counting sort is not an in-place algorithm; even disregarding the count array, it needs separate input and output arrays

Sorted (ascending) Sorted (descending) Not sorted are the same complecety and have the same average time because it work on conistant k &n

\*7

* **Comb Sort**

Pseudocode

**function** combsort(**array** input) **is**

 gap := input.size // Initialize gap size

 shrink := 1.3 // Set the gap shrink factor

 sorted := false

 **loop while** sorted = false

 // Update the gap value for a next comb

 gap := floor(gap / shrink)

 **if** gap ≤ 1 **then**

 gap := 1

 sorted := true // If there are no swaps this pass, we are done

 **end if**

 // A single "comb" over the input list

 i := 0

 **loop while** i + gap < input.size // See [Shell sort](https://en.wikipedia.org/wiki/Shell_sort) for a similar idea

 **if** input[i] > input[i+gap] **then**

 [swap](https://en.wikipedia.org/wiki/Swap_%28computer_science%29)(input[i], input[i+gap])

 sorted := false

 // If this assignment never happens within the loop,

 // then there have been no swaps and the list is sorted.

 **end if**

 i := i + 1

 **end loop**

 **end loop**

**end function**

\*4

it’s improved on bubble sort, bubble sort always compares adjacent values, so it remove and work on one value by one. The basic idea of comb sort is that the gap can be much more than 1. The inner loop of bubble sort, which does the actual *swap*, is modified such that the gap between swapped elements goes down (for each iteration of outer loop) in steps of a "shrink factor".

The pattern of repeated sorting passes with decreasing gaps is similar to Shellsort, but in Shellsort the array is sorted completely each pass before going on to the next-smallest gap. Comb sort's passes do not completely sort the elements. This is the reason that Shellsort gap sequanses have a larger optimal shrink factor of about 2.2

New avrege gap number every cycle

Time Complexity: Average case time complexity of the algorithm is O(N2/2p), where p is the number of increments. The worst-case complexity of this algorithm is O(n2) and the Best Case complexity is O(nlogn).

space complexity: 1

stability: it’s not stable

Comb sort is an in place sorting algorithm: Comb sort does not require auxiliary space for manipulating input so it is an in place sorting algorithm but merge sort does require O(n) of auxiliary space which makes comb sort better in terms of space complexity.

running time:

Sorted (ascending): O(n log n) , as it’s the minimum time as it will check all numbers and not change the place and there is if statement

Sorted (descending): O(n^2), as it will go in the nested loop for all numbers

 Not sorted: O(n^2/2^p) (p is a number of increment), in average as it will break from while loop earler than all number

* **Gnome Sort**

procedure gnomeSort(a[]):

 pos := 0

 while pos < length(a):

 if (pos == 0 or a[pos] >= a[pos-1]):

 pos := pos + 1

 else:

 swap a[pos] and a[pos-1]

 pos := pos – 1

\*5

it works by shifting 2 elements every time starting from the left and compare it with it’s right element and check who’s bigger and sheft them to make the biggest on the right and do this for all elements decreasing for example if there is 4 elements we check the 1st and 2nd then check the 2nd with the 3rd then recheck bigger of the first 2 with the 3rd and so on

time complexity: O(n2) we may think this is a O(n) but we increase and decrease at the same time which results to same as nested loop almost

space complexity: 1

stability: yes it is

in place: Gnome Sort is an in-place sort that is does not require any extra storage

Sorted (ascending): O(n) , as it will increase only witout need to change any element and that means no need to decrese which means it will walk on the elements once only

Sorted (descending): O(n^2), as it will increase and decrease and this is such nested loop were we have to check the numbers again after the first sort for all numbers

 Not sorted: O(n^2it will have some increases and decreases and that would take n2 time as it may cross most of them. \*6

#include <stdio.h>

 void main()

 {

 int i, temp, ar[10], n;

 printf("\nEnter the elements number:");

 scanf("%d", &n);

 printf("\nEnter elements:\n");

 for (i = 0; i < n; i++)

 scanf("%d", &ar[i]);

 i = 0;

 while (i < n)

 {

 if (i == 0 || ar[i - 1] <= ar[i])

 i++;

 else

 {

 temp = ar[i-1];

 ar[i - 1] = ar[i];

 ar[i] = temp;

 i = i - 1;

 }

 }

 for (i = 0;i < n;i++)

 printf("%d\t", ar[i]);

 }

### summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sort method | time complexity | space complexity | stability | in place or not |
| Counting sort | O(n+k) | O(k) | Yes | no |
| Comb sort | O(N2/2p) | 1 | No | yes |
| Gnome sort | O(n2) | 1 | yes | yes |

### references

1 [Counting sort - Wikipedia](https://en.wikipedia.org/wiki/Counting_sort)

2 [Counting Sort - GeeksforGeeks](https://www.geeksforgeeks.org/counting-sort/)

3 [C Program to Sort Array using Counting Sort - Sanfoundry](https://www.sanfoundry.com/c-program-sort-array-using-counting-sort/)

4 [Comb sort - Wikipedia](https://en.wikipedia.org/wiki/Comb_sort)

5 [Gnome sort - Wikipedia](https://en.wikipedia.org/wiki/Gnome_sort)

6 [Gnome Sort (opengenus.org)](https://iq.opengenus.org/gnome-sort/#:~:text=Gnome%20Sort%20is%20an%20in,similar%20to%20a%20bubble%20sort.)

7 [Sorting Algorithms - LAMFO (lamfo-unb.github.io)](https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/)