Cycle Sort

1. worst case → $O(n^2)$
2. Does min memory writes than any other algo (sort).
3. In place and not stable.
4. Useful to solve questions, like, find min swaps needed to sort an array.

Algo.

1) Start with first ele, equal to temp,
2) count = no: of elem smaller than temp.
3) place temp / swap temp with arr [count].
4) Now the ele that was on count will be temp repeat from 2).

→ Code

→
```
void cycleSort( int arr[], int n)
{ for( int i=0; i<n-1; i++)
    { int item = arr[i];
      int pos = i;
      for( int j = i+1; j<n; j++)
      { if (arr [j] < item) { pos++; } }
      swap ( item , arr [pos]);
      while ( pos != i)
      { pos = i;
        for(int j = i+1; j<n; j++)
        { if(arr[j] < item) pos++; }
        swap (item , arr[pos] );
      }
    }
}
```

Heap Sort.

① Like in the case of Selection sort, here also we find max ele, place at last
then sec larg ele, place at sec last.

② But in selection sort, Linear search is used.
In heap sort, max heap data structure.

→ root = max ele., we swap with last ele
→ then heapify
→ Repeat.

O(nlogn).

③
```
Void heapSort (int arr[], int n)
{
    buildHeap(arr,n);  ——→ To be discussed later.
    for (int i=n-1; i>=1, i--)
    {   swap (arr[0], arr[i]);
        heapify (arr, i, 0);
    }
}
```

Counting Sort

① K i.e 0 to K, range of numbers must be given.

→ Not a comparision based algo.

② O(n+k) → when useful. O(n+k) → Space.

If goes above n logn then not useful to us.

→ Stable.

③ Algo

K should be really small.

```
Void CountSort (arr, n, k)
{ int count [k];
  for(int i=0; i<k; i++)
  { count[i]=0; }
  for( int i=0; i<n; i++)
  { count[ arr[i]] ++ ; }   → building arr that counts occurences.

  int index = 0;
  for (int i=0; i<k; i++)
  {  for (int j=0; j< count[i] ; j++)
     {
       arr[index] = i
       index++;
     }
  }
}
```

Radix Sort

- Linear time algorithm if data is in limited range.
  - Stable algo.
- Uses counting sort as a subroutine.
- ∴ Not comparision based.

$O(d * (n + b))$ ← b: base log, d: no: of digits

$O(n + b)$ → space.

Q → {319, 212, 6, 8, 100, 50}

Step 1 :- Rewrite all digits with num of digits in each ele as the no: in largest ele.

{319, 212, 006, 008, 100, 050}

Step 2 :- Sort acc to each digit going from least significant to most significant.

Stable sort

① → 100 050 212 006 008 319

② → 100 006 008 212 319 050

③ → 006 008 050 100 212 319

```
void radixSort(int arr[], int n)
{ int mx = arr[0];
  for(int i=1; i<n; i++){ if(arr[i] > mx)
                            { mx = arr[i] }
  }

  for(int exp=1; mx/exp>0; exp = exp*10)
    { countingSort(arr, n, exp); }
}
```

** Only diff in this countArr will be count[(arr[i]/exp)%10]++;

Instead of count[arr[i]]++;

Bucket Sort

① → In this sort we form slots (or) buckets.

resulting in uniform distribution, to be used effectively.

→ $O(n)$

→ Worst is $O(n^2)$ when all elem in same buckets.

Eg:-

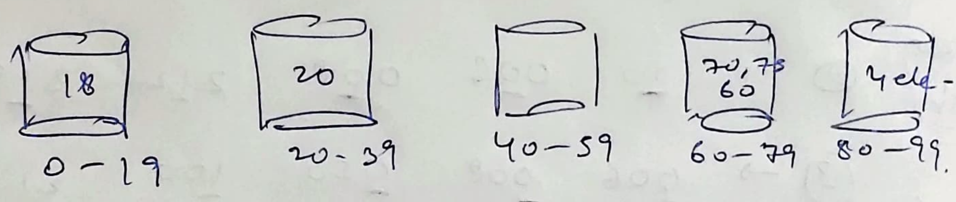
I/p: 20, 88, 70, 85, 75, 95, 18, 82, 60

∴ Range → 0 to 99.

We take 5 buckets (* Taking no: of buckets is a tradeoff b/n time and space).

Step 1,

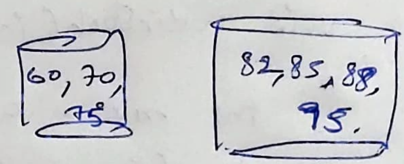
Scatter



Step 2

Sort individual buckets.
(using any sorting algo)

No change in first three



Step 3

Join sorted buckets.

18, 20, 60, 70, 75, 82, 85, 88, 95.

2 pg implementation, refer gfg.