Q1:

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $\boldsymbol{O(n)}$ time.
2. The algorithm is **stable**.
3. The algorithm sorts **in place**, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

As k=1 in this case, so **Counting Sort** will run in $O(n+k) = O(n+2) = \boldsymbol{O(n)}$ time; moreover, Counting Sort is also a **stable** sorting algorithm. This will ***not*** be an *in place* sorting algorithm though.

b. Give an algorithm that satisfies criteria 1 and 3 above.

As there are only two values for the key, ***Partitioning*** around the smaller value will completely sort the array.

```
i=1
while i≤n and A[i].key==1
    i=i+1
if i≤n          //A[i].key==0
    exchange A[i] with A[n]
    x=Partition (A,1,n)
```

As the time complexity of the Partition algorithms is linear, the entire array will be sorted in $\boldsymbol{O(n)}$ time. Moreover, Partition function does not require more than constant amount of memory, hence, this sorting will be **in place** as well. This will ***not*** be a *stable* sorting algorithm though.

c. Give an algorithm that satisfies criteria 2 and 3 above.

Insertion Sort is both a **stable** sorting algorithm, as well as, it is an **in place** sorting algorithm. So it will do the job. This will ***not*** be a *linear time* algorithm though.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

Algorithms qualifying for **part a** can be used as the sorting method in line 2 of RADIX-SORT. The RADIX-SORT will sorts n records with b-bit keys in $O(bn)$ time, as the algorithms are both

- stable sorting algorithms, and
- have $O(n)$ time complexity

Algorithms qualifying for **part b** can ***NOT*** be used as the sorting method in line 2 of RADIX-SORT. The RADIX-SORT will **NOT sort** n records with b-bit keys (regardless of the time complexity), as the algorithms are ***NOT stable*** sorting algorithms.

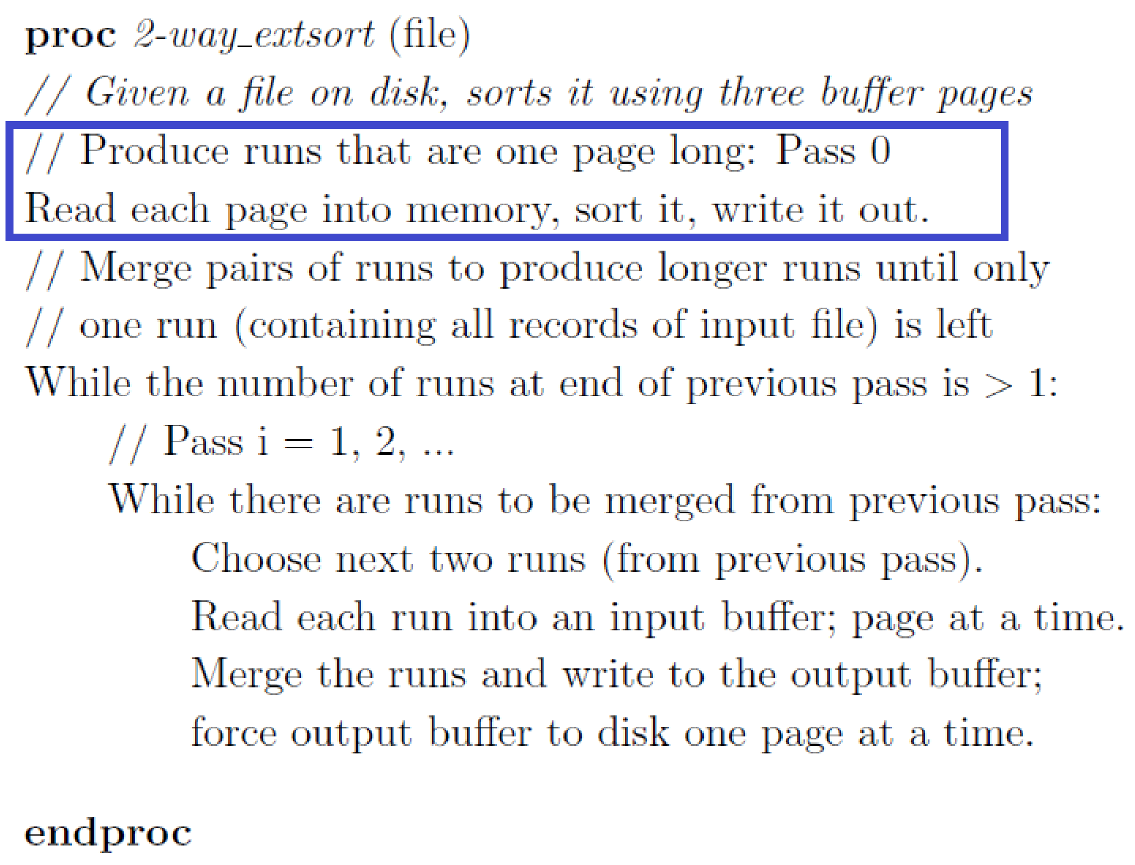
Algorithms qualifying for **part c** can ***NOT*** be used as the sorting method in line 2 of RADIX-SORT. The RADIX-SORT **will sort** n records with b-bit keys but ***not necessarily in*** $\boldsymbol{O(bn)}$ ***time***, as the time complexity of the algorithms qualifying for part c is not necessarily $O(n)$.

Q2:

Suppose that you have a file with 100 pages and that you have three buffer pages. Answer the following questions assuming that a two-way external sort is used:

1. How many runs will you produce in the first pass?

   Let's name the first pass "Pass 0". Using the conservative approach (the one in the pseudo code shared), the algorithm will ***read each page into memory, sort it, and write it out***; thus producing 100 sorted runs of one page each.



```
proc 2-way_extsort (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are one page long: Pass 0
Read each page into memory, sort it, write it out.
// Merge pairs of runs to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1, 2, ...
    While there are runs to be merged from previous pass:
        Choose next two runs (from previous pass).
        Read each run into an input buffer; page at a time.
        Merge the runs and write to the output buffer;
        force output buffer to disk one page at a time.

endproc
```

Two-Way Merge Sort

2. How many passes will it take to sort the file completely?

Pass 1:

Choose two runs (from 100 sorted runs produced in Pass 0). Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 1, for every two sorted runs (of one page each) from the previous pass, we have one sorted run (of two pages); totaling 50 sorted runs.

```
proc 2-way_extsort (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are one page long: Pass 0
Read each page into memory, sort it, write it out.
// Merge pairs of runs to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1, 2, ...
    While there are runs to be merged from previous pass:
        Choose next two runs (from previous pass).
        Read each run into an input buffer; page at a time.
        Merge the runs and write to the output buffer;
        force output buffer to disk one page at a time.
endproc
```

Two-Way Merge Sort

Pass 2:

Choose two runs (from 50 sorted runs produced in Pass 1). Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 2, for every two sorted runs (of two pages each) from the previous pass, we have one sorted run (of four pages); totaling 25 sorted runs.

Pass 3:

Choose two runs (from 25 sorted runs produced in Pass 2). Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 3, for every two sorted runs (of four pages each) from the previous pass, we have one sorted run (of eight pages); totaling 12 sorted runs of eight pages each and one run of four pages (i.e. total 13 runs).

Pass 4:

Choose two runs (from 13 sorted runs produced in Pass 3). Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 4, for every two sorted runs (of eight pages each) from the previous pass, we have one sorted run (of sixteen pages); totaling 6 sorted runs of sixteen pages each and one run of four pages (i.e. total 7 runs).

Pass 5:

Choose two runs (from 7 sorted runs produced in Pass 4). Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 5, for every two sorted runs (of sixteen pages each) from the previous pass, we have one sorted run (of thirty two pages); totaling 3 sorted runs of thirty two pages each and one run of four pages (i.e. total 4 runs).

Pass 6:

Choose two runs (from 4 sorted runs produced in Pass 5). Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 6, we have one sorted run of 64 pages (by merging two sorted runs of 32 pages each), and one sorted run of 36 pages (by merging two sorted runs, one of 32 pages and other of 4 pages).

Pass 7:

Read each run into an input buffer; page at a time. Merge the runs and write to the output buffer; force output buffer to disk one at a time.

At the end of Pass 7, we have **one sorted run** of 100 pages and the file is sorted.

Total Number of Passes:

From Pass 0 to Pass 7, a total of **8 passes** $(\lceil \lg 100 \rceil + 1)$ would be needed to sort the file completely.