CIS 435 Programming Exercise Module 5 (30points)

1 What to turn in

Follow the guidelines of Handout 10 dated January 23, 2004. Submissions that deviate from these guidelines will be assigned 0 points.

2 What to implement

Implementation is required for the function described in part A.

3 Part A: Implementation of Countsort/Radix-Sort (20 points)

The purpose of Part A is to implement a radix-sort based algorithm for sorting 32-bit integers. A 32-bit integer can be viewed as an 1-digit radix-$(2^{32}-1)$ integer, or 2-digits radix-65536 integer or 4-digit radix-256 integer. Provide an ANSI C implementation of a countsort-based algorithm with the following syntax and behavior.

```
void intsort_cnt(unsigned int  *keys, int n, unsigned int mradix);
```

`keys` is a pointer to the input array which consists of positive integers from 0 through $2^{32}-1$. The length of the array (input size) is `n`. Each integer in the range $0 \ldots 2^{32}-1$ is viewed as a d-digit radix-`mradix` integer. Therefore, parameter `mradix` determines how the integers in `keys` are considered by `intsort_cnt` and how your algorithm will operate on input `keys`.

1. If $mradix = 256$, your algorithm should behave like a RadixSort algorithm and view the integers as 4-digit radix-256 numbers,
2. otherwise, if $mradix = 65536$, your algorithm should behave like a RadixSort algorithm and view the integers as 2-digit radix-65536 numbers,
3. otherwise, if $n > mradix$, your algorithm should behave like CountSort,
4. otherwise, if $n < mradix$, then your algorithm should automatically decide the radix among radix-256 and radix-65536 and revert to a radix-sort based algorithm (ie choose between (1) and (2)).

Remark. Note the order of the if-otherwise statement above. If $n = 10000$ and $mradix = 256$, case 1 applies, even if case 3 is also applicable.

`Grade scheme.` Among the 20 points assigned to this problem, 15 points will be given for a correct implementation and 5 points will be given to any solution that decides correctly each one of the four cases above.

4 Part B: Experimental Results (10 points)

You will run your implementations with the testing functions provided in `sortg.c` on 3 different data-sets and 4 problem sizes.

1. **Problem sizes** are
   1. $n = 64000$.
   2. $n = 256000$.

3. $n = 1024000$.

4. $n = 4096000$.

2. The three different **data sets** consist of the following test instances.

   1. An array of integers where all the elements are the same (say n).

   2. A reverse sorted array of integers where the i-th element of the array is $n - i$.

   3. An array whose elements are randomly chosen using function `random` (see `sortg.c` on how to setup such an array).

a) For each problem size and test instance describe in tabular form (Table B1) the running time of your implementation. What was the value of `mradix` in each case? A timing of the execution of any function can be obtained similarly to the one provided in `sortg.c`. (6 points)

b) For *problem sizes*

1. $n = 64000$.

2. $n = 256000$.

3. $n = 1024000$.

4. $n = 4096000$.

and for *mradix=256 and mradix=65536*,
describe in tabular form (Table B2) the running time of your implementation for each of the 8 possible combinations of `n` and `mradix`. (4 points)

Remarks.
The tables to be reported in part B should be included at the end of the submitted source code file as comments. Label them B1 and B2 clearly.