CS 230: Algorithms and Data Structures — Fall 2021
Prof. Darren Strash
Hamilton College

Mini Problem Set 8 — Radix Sort with Bits

Collaboration Statement: Collaboration is allowed, but each student must write their own solution. *Ideas* may be shared but artifacts (such as drafts, solutions, code, or pseudocode) *must not be shared*. For written problems, you are allowed to reach a solution collaboratively with others on a whiteboard. Any solutions reached collaboratively in this way *must be destroyed* before writing your final solution on your own, separated from your collaborator(s). List your collaborators, as well as the nature of your collaboration on your submission. Each student's submission is graded *individually*. For coding problems, you are allowed to discuss and brainstorm ideas at a high level, but you are not allowed to look at each other's code.

Turn in your assignment: Turn in your assignment by submitting your written solutions and code on Gradescope by **Tuesday, 11/09 at 11pm**, with a grace period (a free extension) up to **Thursday 11/11 at 11pm**. You should submit all of your code files to the assignment labeled with `(code)`. The LaTeX source of this document is available at `http://tiny.cc/cs230-latex`, in case you wish to use it for your submission.

(Implementing Radix Sort with Bits, 10pts.) Accept the following GitHub assignment.

`https://classroom.github.com/a/UbeRexD4`

To compile and test your code, run: `cmake ./` (once), then `make` and `./test_radixsort`.

In this problem, you will implement a more involved radix sort algorithm for `unsigned int`s. Use the following as a guide to help you implement radix sort.

1. A critical tool for radix sort is the ability to extract "digits" in certain positions from a given value to produce a key. Here, instead of extracting one 10-based digit at a time, we will extract bits using bit shifting.

Continued on next page →

Although your algorithm works on `unsigned int`s, examples here are done on 8-bit values. Consider the 8-bit unsigned integer:

$$x = 0011\ 1010_2$$

where we name the bits in x as follows:

$$x = x_7x_6x_5x_4\ x_3x_2x_1x_0.$$

Using only bit shifting (`<<` and `>>`):

(a) give a formula to compute the value of the least significant bit, x_0 from x. (In the example, this is 0_2 which has value 0.)

(b) Now give a formula that will compute the value of the 2 least significant bits, x_1x_0 from x. (In the example, this is 10_2, which has value 2.)

(c) Now give a formula that will compute the value of the bits x_5x_4 (In the example, this is 11_2, which has value 3.)

(d) Generalize your formula from i, ii, and iii as follows: given the number of bits you want to extract ℓ, and the index i of the least significant bit to extract, compute the value of the bits $x_{i+\ell-1} \cdots x_{i+1}x_i$.

2. Now suppose we wish to sort `unsigned int` values using the values of the bits extracted in the manner above. In the file `radixsort.cpp`, implement your formula from (d) in the function `get_key(value, num_bits, start_bit)` and return the value of the sequence of bits from the unsigned integer `value` to be used in the current round of radix sort. Your function `get_key` should run in $\Theta(1)$ time.

3. While not for credit, feel free to make unit tests for your function in `./test_radixsort`. When testing your function, carefully consider which `value` to use.

4. Use the following pseudocode to implement a version of counting sort using your helper function `get_key` in

   `counting_sort(array, num_bits, start_bit)`

   Your implementation of counting sort should put the elements in order according to the bits extracted from each value.

Algorithm 1 Counting Sort.

proc COUNTING-SORT($A[1..n]$, num_bits, start_bit)

1: $k = 2^{\text{num\_bits}} - 1$ // Can be computed in $\Theta(1)$ time with `<<` and `-`
2: $C[0..k] = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots, 0 \rangle$
3: **for** $i = 1$ **to** n
4: key = KEY($A[i]$, num_bits, start_bit) // the key from $A[i]$ that we are sorting on
5: $C[\text{key}] = C[\text{key}] + 1$
6:
7: **for** $i = 1$ **to** k
8: $C[i] = C[i-1] + C[i]$
9:
10: $B[1..n]$
11: **for** $i = n$ **downto** 1
12: key = KEY($A[i]$, num_bits, start_bit) // the key from $A[i]$ that we are sorting on
13: $B[C[\text{key}]] = A[i]$
14: $C[\text{key}] = C[\text{key}] - 1$
15: $A = B$

   Be sure to implement your algorithm to run in $\Theta(n)$ time.

5. While not for credit, feel free to make unit tests for your function in `./test_radixsort`. When testing your function, consider which values to sort so that you are sure the algorithm is correct and stable.

6. Finally, implement `radix_sort(array, num_bits)` which makes calls to `counting_sort` to sort the array one key at a time (where each key has `num_bits` bits) until all elements are sorted.

   For this assignment, you can assume that `num_bits` is either 1, 2, 4, 8 or 16.

7. While not for credit, feel free to make unit tests for your function in `./test_radixsort`.

8. Run `./time_radixsort`, which runs `radix_sort` on random vectors of increasing size, with 1-, 2-, 4-, 8-, and 16-bit keys. If the output is too wide for your screen, you can select which variants to run by placing any of `1-bit`, `2-bit`, `4-bit`, `8-bit`, or `16-bit` as command-line arguments in any order. For example: `./time_radixsort 1-bit 16-bit`.

   Does the output of `./time_radixsort` indicate that your implementation of radix sort has running time $\Theta(n)$? Explain why. Adjust your algorithm so that it has running time $\Theta(n)$.

9. From your experimental results, how many bits should we use for our keys in radix sort?

A contest.

Radix Royale: The authors of the three fastest radix sort implementations earn the title of **Radix Royale**, and a drink of their choice from Fojo. For this contest, you are allowed to consult any resources you wish (however, I'm off limits), but you must write the code yourself and cite your sources. To participate, write your fastest implementation of radix sort in the function `radix_royale(vector)`. To add your implementation to the timing output, run `./time_radixsort royale`.

All submissions will be compared on a variety of inputs in a head-to-head contest on `decker`. Speed is measured by the running time averaged over 5 runs on all inputs. In case of a tie, the student submitting their solution the earliest wins. Entries that fail to sort even one input are disqualified from the contest.

Note the following restrictions: you aren't allowed to build your code with any different compiler or linker flags (so no modifying `CMakeLists.txt` or `Makefile`). I am not available for advice for implementing your submission for the contest.