# Computer Science

Dr. Y. Liow (February 17, 2023)

# Contents

**104 Distribution sort** **4801**
104.1 Distribution sort: counting sort . . . . . 4802
104.2 LSD radix sort . . . . . 4805
104.3 MSD radix sort . . . . . 4810
104.4 Bucket sort . . . . . 4812

File: chap.tex

# Chapter 104

# Distribution sort

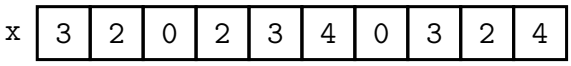
File: counting-sort.tex

## 104.1 Distribution sort: counting sort

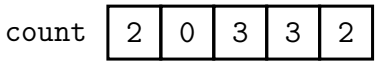
Beside sorting an array using a comparison-based sorting algorithm, i.e., using $\leq$ or $<$ or $\geq$ or $>$ to compare values in the array, there's another class of sorting algorithms that work in a difference way. Distribution sorting algoroithms sort by *distributing* the values in an array. Sometimes distribution sort will use a comparison-based sort.

The simplest distribution sort is the counting sort.

Suppose you have an array `x` of integers with values from 0 to 4 (inclusive):

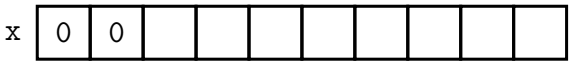



You scan the array and count the number of times each value from 0 to 4 occurs in `x`.

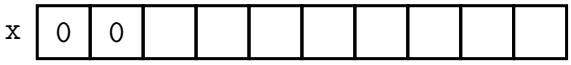



For instance `count[2]` is the number of times `2` occurs in array `x`.

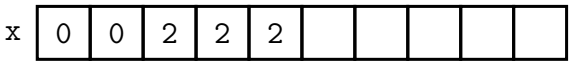
Now the final thing to do is to fill `x` using information in `count`. For instance `count[0]` is `2`, so I would put two `0`'s into `x`:




Next, since `count[1]` is `0`, `1` does not appear in `x`. So the re-organized `x` looks like this (i.e., no change):




However note that `count[2]` is `3`, which means that `2` appears `3` times. Therefore my `x` now look like this:




You get the point.

The pseudocode looks like this. I assume that the values in `x` are from 0 to `M`.

```
ALGORITHM: counting_sort
INPUT: array x of size n

    count = int array of size M + 1, initialized with 0s
    for i = 0, 1, 2, ..., n - 1:
        count[x[i]] = count[x[i]] + 1
    j = 0
    for i = 0, 1, 2, ..., M:
        for k = 1, ..., count[i]:
            x[j] = i
            j = j + 1
```

If `M` is not known, you have to run through `x` to figure that out.

Here's the runtime analysis. The time to create `count`, if it's in the heap, might very well be $O(1)$ (or the memory allocation might fail because of lack of available memory). So that's hard to say. So let's just assume memory allocation for `count` is negligible. First you have to set the `count` array to zeroes: that takes $A(M+1)$ ($A$ is a constant). You need to run through `x` to fill `count` with values: that takes a time of $Bn$. You will to put values back into `x` using `count`. That requires touching each cell in `x`. So for this part, the runtime is $Cn$. Altogether, the runtime is

$$O(n+M)$$

Of course the space requirement is

$$O(M+1) = O(M)$$

**Exercise 104.1.1.** How would you sort a huge array if the values are integer in the interval $[-10, 10]$? □

**Exercise 104.1.2.** How would you sort a huge array if the values are 1000000, 1100000, 1200000, 1300000, ..., 1900000. □

**Exercise 104.1.3.** Modify the counting sort algorithm to sort a huge array of `double`s where there is a (known) small number of distinct `double` values in the array. Say the size of the array is $10^6$ and there are at most 1000 distinct `double`s in the array. What if the values are not known ahead of time? (The idea should also work for other types.) □

**Exercise 104.1.4.** Take a look at the sieve of Eratosthenes and you'll see that it's related to the counting sort. □

File: radix.tex

## 104.2 LSD radix sort

There are two basic variants of radix sort. First I'll talk about the LSD (least significant digit) radix sort.

Suppose you have the following array (I'm going to draw it vertically):

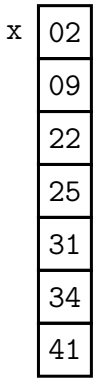
| x |
|---|
| 31 |
| 22 |
| 02 |
| 25 |
| 34 |
| 41 |
| 09 |

First we sort the array *by the ones digit* using a *stable* sorting algorithm (example: mergesort).

| x |
|---|
| 31 |
| 41 |
| 22 |
| 02 |
| 34 |
| 25 |
| 09 |

Because the sorting is stable, note for instance that 22 appear before 02 before and after the sort.

Next I'm going to perform a stable sort on the tens digit:




Note for each pass of the radix sort, you can use any stable sort. You want to pick the fastest (of course): so of course mergesort is better than bubblesort! If you are using mergesort to sort the ones and then the tens, then the runtime is

$$O(2n \lg n) = O(n \lg n)$$

If there are $d$ digits (instead of 2), then the runtime is

$$O(dn \lg n)$$

If most cases, the $d$ might be a constant (i.e., does not change). For instance if your array is an array of integers in a 32-bit computer, then $d = 10$. In this case the runtime is

$$O(dn \lg n) = O(n \ln n)$$

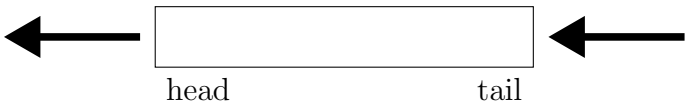
For radix sort, when you look at the key `31` (the first value of the original array) and you are sorting based on the first digit of `31`, i.e., `1`, we say that `1` is the **subkey**. If there's no digit there, for instance look at the key `02` which is actually `2`, then the second digit subkey of `2` = `02` is `0`, i.e., the smallest possible subkey value (if you sorting in ascending order).

subkey

For implementation, you might want to have a `subkey` function. `subkey(key, i)` returns the $i$–th digit of `key` where $i$ starts with 0 (the leftmost digit). If you call `subkey(31, 0)`, you get `1`. If you call `subkey(31, 1)`, you get `3`. Remember that since `2` has only one digit, if you call `subkey(2, 1)`, you get `0`.

There's another way to do this with a better runtime.

Recall the concept of a queue, a data structure there data enters at the tail of the data structure and leave through the head of the data structure:




We saw the queue in CISS245 where we implemented it using an array (say dynamic arrays in the heap). The worst runtimes are insert and delete are

1. entering the queue: $T(n) = O(n)$
2. leaving the queue: $T(n) = O(n)$

where $n$ is the size of the queue.

There's a better way to implement a queue ...

After we study linked list, you will see that you can implement a queue using linked lists. The operations on a queue implemented using a linked list has the following runtimes:

1. entering the queue: $T(n) = O(1)$
2. leaving the queue: $T(n) = O(1)$

where $n$ is the size of the queue.

We then sort based on the digit value in the following way: First we create an array `q` of 10 queues. `q[0]` is a queue that will hold values with digit value of `0`, `q[1]` is a queue that will hold values with digit value of `1`, etc. For each value of `x`, we place the value into the appropriate queue based on the digit of that value. When we're done, we remove all the values in `q[0]` and place them in order into `x`. We then repeat with `q[1]`, etc. Because the operations on the queue all have runtime of $O(1)$, the total runtime is $O(n)$.

We then repeat the process, but now determine which queue a value of `x` should go into by the tens digit of that value. Altogether the runtime is $O(2n)$.

In our example above, the numbers have 2 digits. If the numbers have $d$ digits, then the runtime is $O(dn)$. If the number of digits $d$ is fixed (i.e., is a constant), then the runtime is $O(n)$.

For space, the total amount of memory used by the queues adds up to $O(n)$. So although the queue-based version of radix sort is faster, it's at the expense of memory. Remember that. Nothing is for free.

Note that by using queues, the sorting process is stable: a number that has joined a queue will always be ahead of another that joined the queue later. After each pass, the values of a queues are dequeued and put back into the original array, of course starting with the queue corresonding to `0`.

Note that when we use queues for distribution, a value enters the queue without being compared when other values. Therefore in this context radix sort is a distribution (non-comparison) sorting algorithm.

Another thing to note is that we are assuming all the numbers we are sorting have the same number of digits. What if the number of digits are different? As mentioned above, if you call `subkey(key, i)` and `key` has no digit at `i`, `subkey(key, i)` returns the smallest possible digit value, i.e., `0`.

The above starts with the "smallest" digit (i.e., the ones digit). That's why it's called **least signicant digit (LSD) radix sort**.

least signicant digit (LSD) radix sort

**Exercise 104.2.1.** Perform LSD radix sort on the following values, showing the values of the array after each pass:

$$121, 23, 231, 89, 123, 536, 697, 457, 355, 243, 657, 345, 174, 248, 346, 376$$

□

**Exercise 104.2.2.** What happens when you sort not by single digit for each pass, but *two* digits? □

## 104.3 MSD radix sort

Instead of sorting numbers by sorting on their digits from right to left, i.e., starting with the least significant digit, you can do the same thing but going left to right. That's **MSD radix sort** (MSD = most significant digit). If you sort numbers using MSD radix sort on

MSD radix sort

$$22, 19, 31, 91, 90, 10, 45$$

after the first pass (i.e., after stable sorting on the tens), you you will see this:

$$19, 10, 22, 31, 45, 91, 90$$

Now we (stable) sort by the ones:

$$10, 90, 31, 91, 22, 45, 19$$

For numbers with different lengths, you can pad on the right with an extra "digit" that is infinitely small. For instance, say you pad the above on the right with `?` and assume that $? < 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9$ to get

$$2?, 1?, 3?, 91, 9?, 19, 45$$

Then after performing a stable sorting on the leftmost digit, I get

$$1?, 19, 2?, 3?, 45, 91, 9?,$$

and on after the second pass of stable sorting on the second digit, I get

$$1?, 19, 2?, 3?, 45, 9?, 91$$

After the array is sorted, you can remove this extra data `?` to get

$$1, 19, 2, 3, 45, 9, 91$$

If you think that's bizarre, then think again – you have seen this before. Let's look at strings. Suppose you have an array of string `"apple"`, `"abe"`, `"cat"`, `"ab"`, `"bee"`, `"cap"`. Using MSD radix sort, you would get

```
"ab", "abe", "apple", "bee", "cap".  "cat"
```

where the order of each character is determined by its ASCII code.

The above ordering is called **dictionary order**. Duh.

Note that you don't really need to "pad on the right". For instance when comparing strings using the dictionary ordering on C-strings $x$ and $y$ to compute the boolean value of $x < y$, you run a loop comparing `x[i]` and `y[i]`. If `i` is less than the length of `x` but $\geq$ the length of `y`, you return true. If `i` is less than the length of `y` but $\geq$ the length of `x`, you return false. If `x[i]` $<$ `y[i]`, you return true. If `x[i]` $>$ `y[i]`, you return false. If `x[i] == y[i]`, you increment `i` and continue on to the next iteration of the loop.

File: bucket-sort.tex

## 104.4 Bucket sort

The bucket sort is very similar to the radix sort.

Recall that when I was talking about the radix sort, I use 10 linked lists as queues to sort integers in each pass of the radix sort. I'm going to do it again, but in a different way.

It's really pretty easy. Suppose I know that my arrays always have values from 0 to 99. Then I can create 10 bucket. I scan the array and if a value of the element has value from 0 to 9, I put it into bucket 0. If the value is from 10 to 19, I put it into bucket has 1. Etc. If the number of values is a bucket is small, I can use any suitable sorting algorithm to sort values in the bucket. If there are too many values, of course I can use a suitable sorting algorithm, including the bucket sorting algorithm, So suppose for my first bucket that holds values 0 to 9, I have 1000. Since there are only 10 possible values, I would choose the counting sort and sort the bucket. That's it.

But what if the values are from 0 to 999999999? I can have have 10 buckets again and if the range of values of the buckets are the same, then the first bucket would hold values from 0 to 99999999. Unfortunately, it might not be a good idea to use the counting sort. You have to decide on what other sorting algorithm to use. If you use bucket sort again, then, of course each bucket this time will have a smaller range of values. If you do this recursively, of course, at some point, the number of values in a bucket will shrink – except for the case when there's clustering, i.e., there are lots of values clustered together.

Assume the values are uniformly distributed and you want to minimize memory usage, you can use a linked list (see chapter on linked list) for each bucket. The space complexity is minimal when compared to using vectors. Note that insertion sort can be performed on linked lists (the buckets in this case). Then the values from the buckets can be copied back to the original input array.

# Index

dictionary order, 4810

least signicant digit (LSD) radix sort, 4808

MSD radix sort, 4810

subkey, 4806