CS61B Lecture 29

Monday, April 6, 2020

So far, we have just looked at the given complexity of certain algorithms. There is another question we can ask: for a given problem, what is the lowest worst-case time we can have for a solution? In other words, we are attempting to guarantee a lower bound for a set of solutions to a problem, which is generally harder to come up with than an upper bound.

There's an obvious guess for a lower bound: linear time, since you must at the very least read in all the data, but it's not particularly helpful in this case. Let's work methodically.

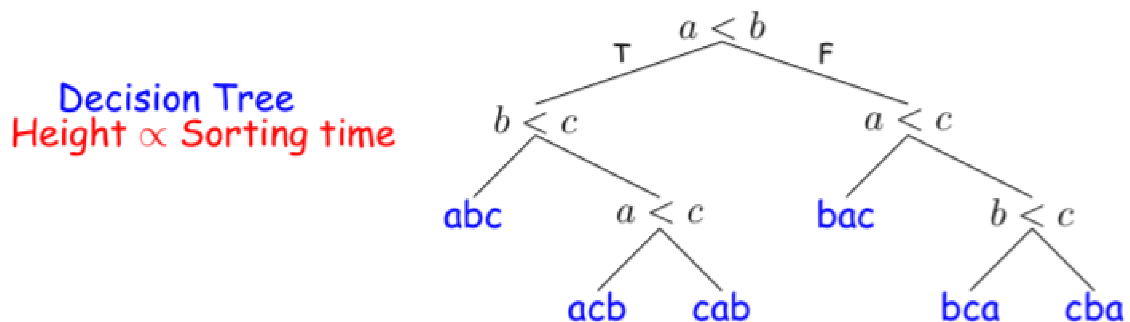
Faster than Linearithmic

We can prove that if the only operation your program is allowed to do is to take two elements and compare them and move the data around, then:

$$\Omega(NlgN)$$

is the best worst-case time you can get out of any algorithm. This is the tricky part: with lower-bound arguments, you must always specify precisely what the algorithms are allowed to do, so that no one can cheat the specifications.

Here's the reasoning behind this lower bound argument: there are N! possible ways the data could be scrambled. Therefore, our program must be set up in such a way so that it can do N! different combinations of data-moving operations, and there must thus be N! possible combinations of outcomes of all the if-tests in your program (since those determine what move gets moved where, assuming all comparisons are 2-way).



Can we put a bound of the height? Since each if-tests goes two ways, the number of possible outcomes for `k` if-tests is

$$2^k$$

Thus, we need enough tests such that:

$$2^K \geq N!$$

which means

$$k \in \Omega(lgN!)$$

Using Stirling's approximation, which allows us to get an approximation of a factorial,

$$N! \in \sqrt{2\pi N}(\frac{N}{e})^N(1+\Theta(\frac{1}{N})),$$

$$lg(N!) = \frac{1}{2(lg(2\pi)+lgN)} + NlgN - Nlge + lg(1+\Theta(\frac{1}{N}))$$
$$= \Theta(NlgN)$$

This tells us that `k`, the worst-case number of tests needed to sort N items by comparison sorting is in

$$\Omega(NlgN)$$

because there must be cases where we need (some multiple of)

$$NlgN$$

comparisons to sort N things.

Distribution

Suppose we can do more than compare keys. For example, how can we sort a set of N integer keys whose values range from 0 to kN, for some small constant k?

One technique is distribution sorting:

- We put the integers into N buckets; integer p goes to bucket

$$\lfloor\frac{p}{k}\rfloor$$

- At most, there will be `k` keys per bucket, so we catenate and use insertion sort, which will now be fast.

For example, with k = 2 and N= 10,

```
Start:
  14  3   10  13  4   2   19  17  0   9
In buckets:
  | 0 | 3  2 |  4 |   |  9  | 10 | 13 | 14 |  17 |  19  |
```

Now insertion sort is much faster. Putting things in buckets takes

$$\Theta(N)$$

and insertion sort takes

$$\Theta(kN)$$

Thus, when `k` is a fixed constant, then we have sorting in time

$$\Theta(N)$$

Distribution Counting

Here's a very similar technique for a different problem: count the number of items less than some value.

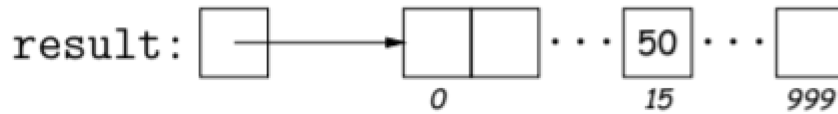
If

$$M_p = \text{number of items with value } < p$$

then in sorted order, the `j`-th item with value `p` must be item

$$M_p + j$$

For example, if we have a set of numbers in the range [0, 1000), and that exactly 15 of them are less than 50, then the result of sorting will look like this:



In other words, the count of numbers less than `k` gives the index `k` in the output array.

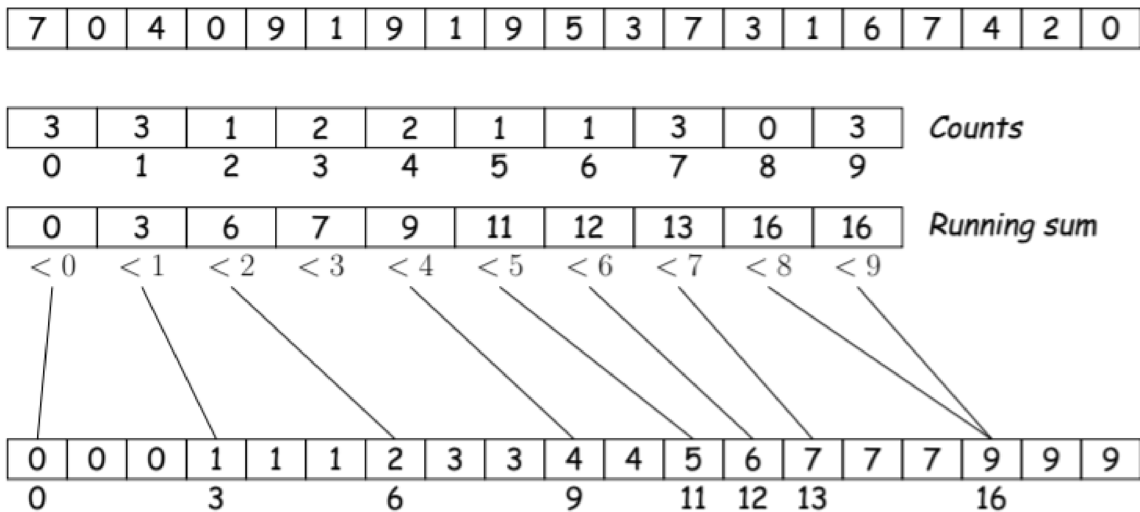
If there are `N` items in the range 0 to `M`-1, it gives another linear time algorithm

$$\Theta(M + N)$$

(including both `M` and `N` to allow for both duplicates and cases where `M` >> `N`).

Example

Suppose all items are between 0 and 9 as in this example:



- The "counts" line gives the number of occurrences for each key.
- The "running sum" gives the cumulative count of keys that are less than each value, which tells us where to put each key:
- The first instance of key `k` goes into slot `m`, where `m` is the number of key instances that are less than `k`.

Here's another example:

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

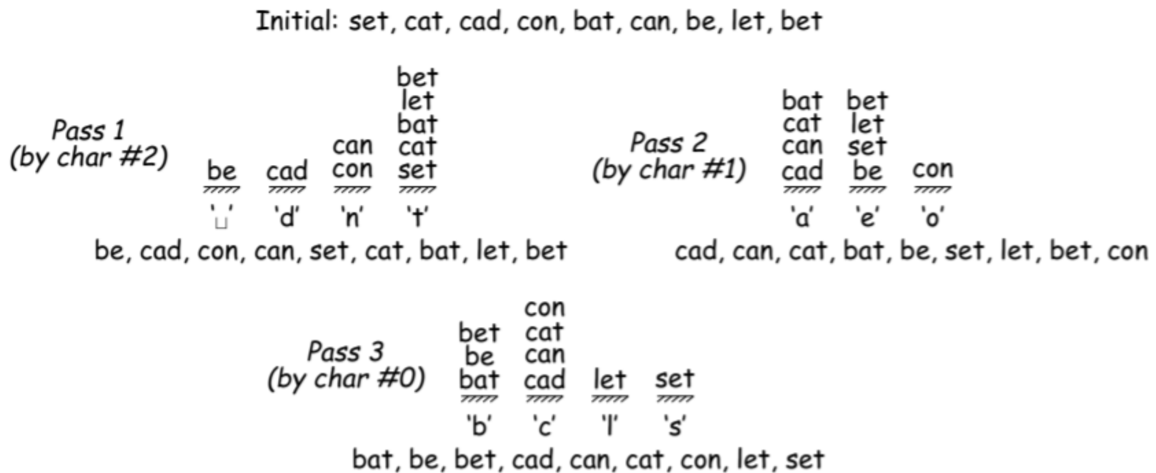
	0	1	2	3	4	5	6	7	8	9
Counts	3	3	1	2	2	1	1	3	0	3
Running sum of Counts	0	3	6	7	9	11	12	13	16	16
Next positions	3	6	7	9	11	12	13	16	16	19

	0			3			6			9			12			15			18
Output	0	0	0	1	1	1	2	3	3	4	4	5	6	7	7	7	9	9	9

Radix Sort

The idea behind radix sort (an old one) is to sort keys one character at a time. We can use distribution counting for each digit, and we can either work right to left (LSD radix sort, where LSD is short for least significant digit), or left to right (MSD radix sort).

LSD radix sort is venerable, and can be used for punched cards:



The reason we do it from behind is that this sorting is stable, and because we recursively call the algorithm, we are guaranteed that at each step after the first, any character after the one we are sorting by is in the order we want it in.

MSD Radix Sort

MSD radix sort is a bit more complicated because we must keep lists from each step separate. However, it has the benefit of avoiding processing one element lists (let and set do not need to be processed after the first in the example below):

A	posn
⋆ set, cat, cad, con, bat, can, be, let, bet	0
⋆ bat, be, bet / cat, cad, con, can / let / set	1
bat / ⋆ be, bet / cat, cad, con, can / let / set	2
bat / be / bet / ⋆ cat, cad, con, can / let / set	1
bat / be / bet / ⋆ cat, cad, can / con / let / set	2
bat / be / bet / cad / can / cat / con / let / set	

The * indicates which bin is being sorted at that step.

Performance of Radix Sort

Radix sort takes

$$\Theta(B)$$

where `B` is the total size of the key data, which means it is different from every algorithm we have looked so far, which are proportional to the number of keys/records.

How can we compare this algorithm to the others then?

To have N different records, we must have at least

$$\Theta(lgN)$$

long, because that's how many characters you need to have at least N different keys. For example, if you have a 26-character alphabet, the size of a word, given there are

$$26^2$$

words, there has to be at least two characters to distinguish them all from that alphabet.

In other words:

$$lg_{26}x \in \Theta(lg_2x)$$

Furthermore, comparison actually takes time

$$\Theta(k)$$

where `K` is the size of the key in the worst case (think about how we compare strings to each other).

That means the

$$NlgN$$

comparisons really mean

$$N(lgN)^2$$

operations.

While radix sort would take

$$B = NlgN$$

time with minimal-length keys, we must also work to get good constant factors with radix sort.

Search Trees

A search tree is in sorted order, when read in inorder.

- Need balance to really use for sorting [next topic].
- Given balance, same performance as heapsort: N insertions in time of lg N each, plus linear time to traverse, gives:

$$\Theta(N + NlgN) = \Theta(NlgN)$$

Summary

- Insertion sort:

  $$\Theta(Nk)$$

  comparisons and moves, where `k` is the maximum amount of data displaced from final position, which is good for small datasets, or almost ordered data sets.
- Quicksort:

  $$\Theta(NlgN)$$

  with good constant factors if the data is not "pathological" (designed specifically to cause problems for the algorithm, e.g. intentionally causing it to always pick the wrong pivot). Worst case of

  $$\Theta(N^2)$$

- Merge sort:

  $$\Theta(NlgN)$$

  guaranteed. Good for external sorting.
- Heapsort/tree sort with guaranteed balance:

  $$\Theta(NlgN)$$

  guaranteed.
- Radix sort, distribution sort:

  $$\Theta(B)$$

  (number of bytes). Also good for external sorting.