Sorting #5 - Final quicksort + Counting + Radix

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

- Announcements
- Thing of the day

Announcements

- Next MA MUST go out after class
- I'm so very done with this week, it was a long one!

Thing of the day: Waterfall Printer

Quicksort - Named (mostly) appropriately

- Runs in O(N log N) time
- Uses a divide and conquer strategy (like merge sort)
- Very sensitive to implementation
  - Pick a good pivot or you're doomed
- Recursive algorithm
  - Can work in the single array, so no doubling of memory space

Performance of Quicksort

- Average running time: O(N log N)
- Worst case: O(N^2)
  - Can be made exponentially unlikely with a small tweak
  - All of the performance issues are centered around picking the pivot
- Often combined with other sorts to improve efficiency:
  - When N (of any sublist) is 5 <= N <= 20 -> change to Insertion sort

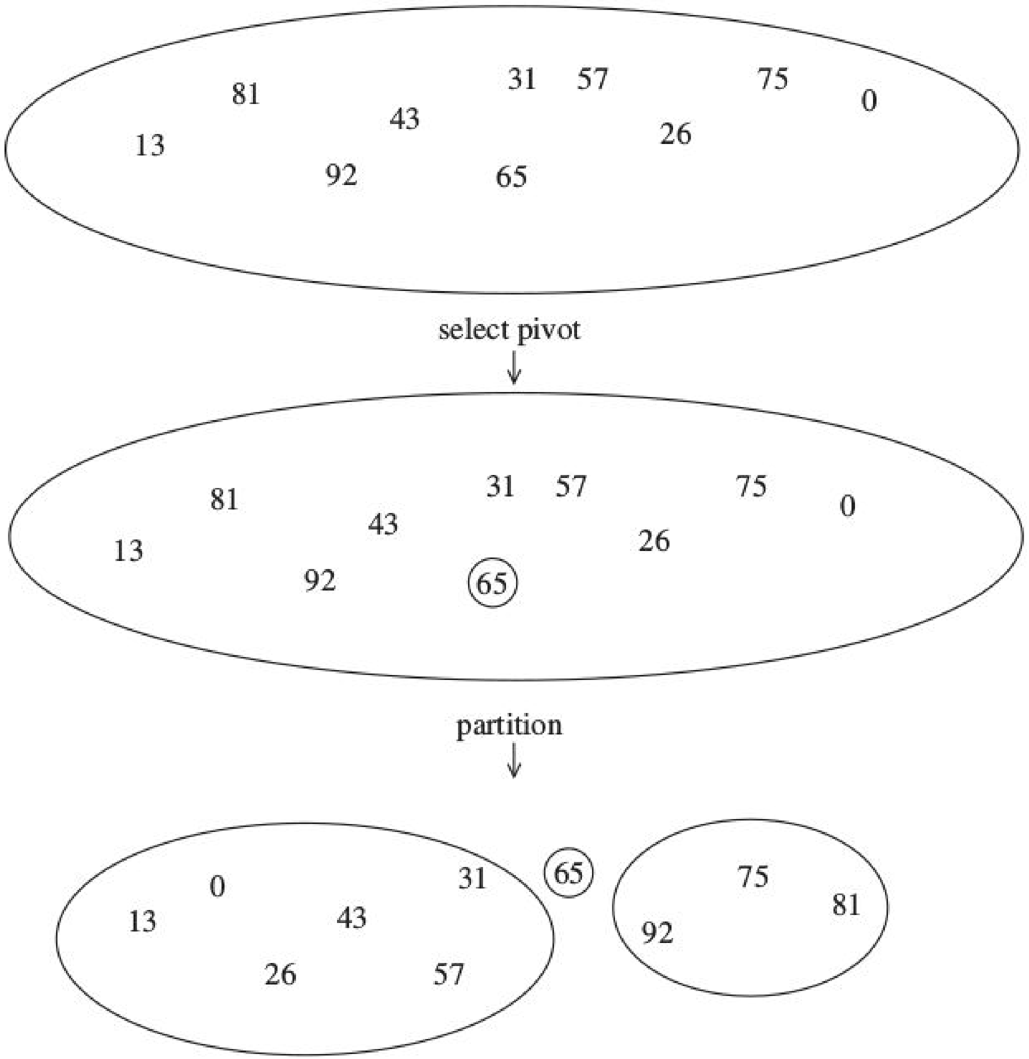
Basic Idea: Divide over pivot, repeat

- For list S:
  - If S.size() is 0 or 1, return
  - Pick element v in S, name it pivot
  - Partition S - {v} (take out the pivot) into two groups:
    - Those smaller than v -> S_1
    - Those bigger than v -> S_2
  - Return {quicksort(S_1), v, quicksort(S_2)}

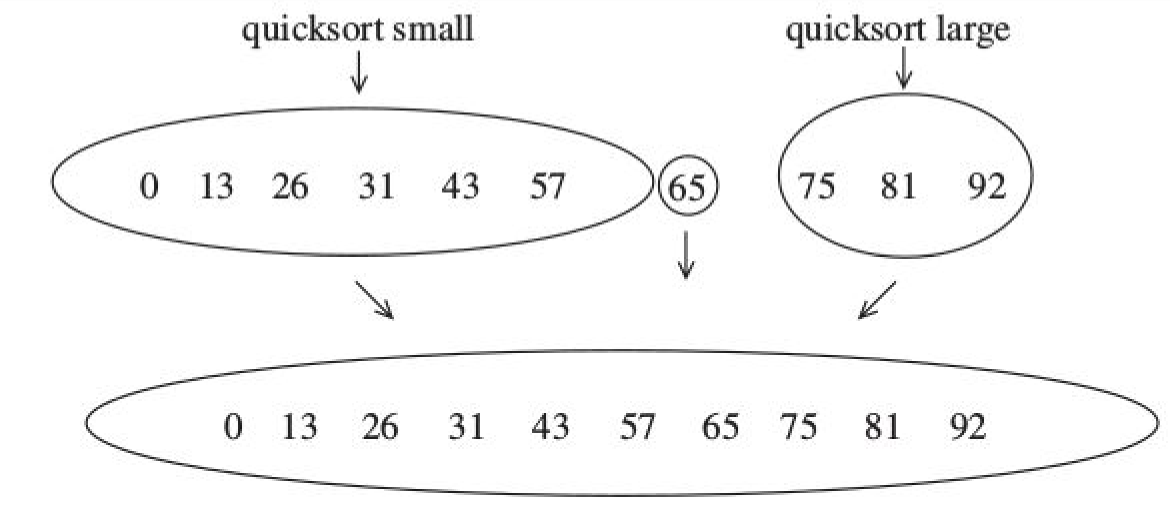
WIN

Divide over pivot

- Pick pivot
- Divide into two lists
  - S_1 -> Those < pivot
  - S_2 -> Those > pivot



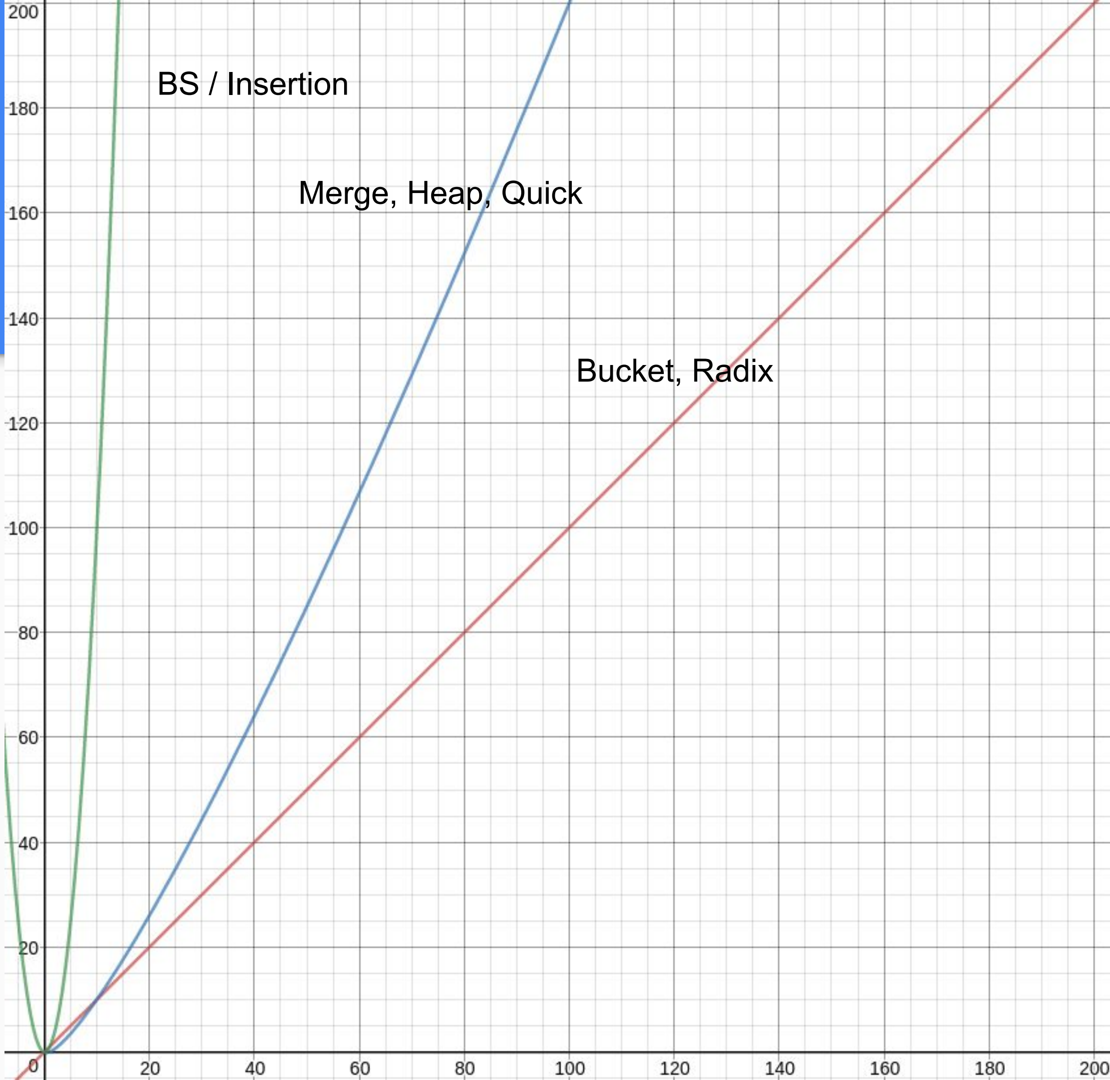
Combine back together



```
template <typename Comparable>
void SORT( vector<Comparable> & items )
{
    if( items.size( ) > 1 )
    {
        vector<Comparable> smaller;
        vector<Comparable> same;
        vector<Comparable> larger;

        auto chosenItem = items[ items.size( ) / 2 ];

        for( auto & i : items )
        {
            if( i < chosenItem )
                smaller.push_back( std::move( i ) );
            else if( chosenItem < i )
                larger.push_back( std::move( i ) );
            else
                same.push_back( std::move( i ) );
        }

        SORT( smaller );     // Recursive call!
        SORT( larger );      // Recursive call!

        std::move( begin( smaller ), end( smaller ), begin( items ) );
        std::move( begin( same ), end( same ), begin( items ) + smaller.size( ) );
        std::move( begin( larger ), end( larger ), end( items ) - larger.size( ) );
    }
}
```

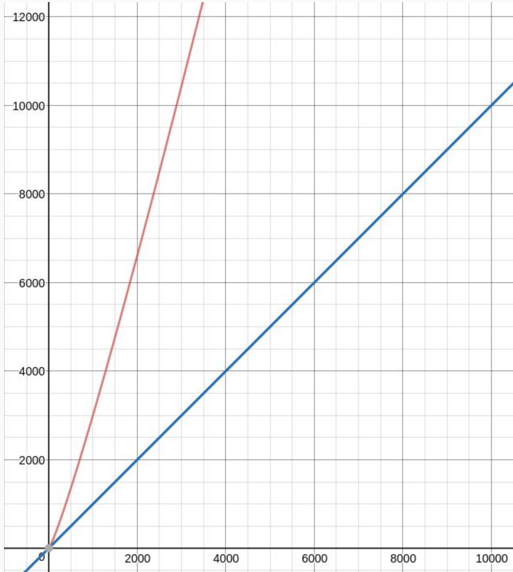
Linear Time Sorting

- Counting / Bucket sort
- Radix sort



Linear Time Sorts

- If your data has certain features, there's linear time sorting algorithms
  - O(N) << O(N log N)
- Normally, this is exploiting size or ordering of data
- These are not "general purpose" sorting algorithms
  - Use some kind of non-comparison approach
- Ones in book:
  - Bucket (Counting) sort
  - Radix sort



Bucket (Counting) sort

- Linear time sort for small integers
- Given: Arr = [A_1, A_2, … A_N] of positive integers smaller than M.
- Make int counts[M], initialized to all 0.
- Read input: for each value in Arr you increment counts with: counts[A_i]++
- Then print out all non-zero buckets counts[i] times
- Results are found in O(M+N) time
  - If M == O(N), then final result is O(N)
- Power comes through M-way comparisons in O(1) time
  - Similar to hashing algorithm

How useful is bucket sort?

- Requires having only positive integers
- Requires a known maximum value
- Requires an array of integers[maximum value] of storage space:
  O(maxValue in array to sort)
- Does this perfect storm of features happen often?

<u>YES</u>

Consider some use cases

- Ages of people
- Social security number sorting: XXX-XX-XXXX -> counts[1000000000]
    - Too big? Unsure
- Car mileages in a fleet
- Could combine with a hash table, hashed by sorted key to lookup data?
    - Best of both worlds! Boom!
- Calculating medians of data sets

Different approach: Radix sort

- Historically used to sort punch cards, so also called card sort
- Can work on integers or strings
- Sorts by one indexed position at a time
    - Done right to left (won’t work otherwise!) - must got LSB to MSB
- Is a stable sort
- Effectively is a multi-pass bucket sort

Radix sort description

- Data must be N strings/values in range(0..b^p-1)
  - N == set of members in array
  - b == number of buckets, which is determined by the size of the alphabet in the data
  - p == number of passes, which is the max length of the strings/numbers
- Do passes over each digit or string position
  - Go from least significant to most significant
  - Put item into bucket as indexed by position (needs b buckets)
  - When done, read all items out in FIFO (to be a stable sort) order back to original array
  - Repeat p times, moving index from least to most significant position
- Result is done in O(p(N+b)) time!

Counting radix sort - sorting by numbers

INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Each pass orders the items in a more significant way.

The results kind of pop into existence at the end.

Book code

- Not complex
- Requires the right *kind* of data set to be useful

```
void radixSortA( vector<string> & arr, int stringLen )
{
    const int BUCKETS = 256;
    vector<vector<string>> buckets( BUCKETS );

    for( int pos = stringLen - 1; pos >= 0; --pos )
    {
        for( string & s : arr )
            buckets[ s[ pos ] ].push_back( std::move( s ) );

        int idx = 0;
        for( auto & thisBucket : buckets )
        {
            for( string & s : thisBucket )
                arr[ idx++ ] = std::move( s );

            thisBucket.clear( );
        }
    }
}
```

What limits and why isn't it everywhere?

- Why doesn't this get used for long strings?
  - The key is in how string comparisons are done in programming languages
  - Also, has to handle a set with varying length strings (consider zero padded)
- Why doesn't this get used for large alphabets
  - How many total Unicode characters are there?
  - Book uses the extended ASCII set of 256, but…
    - Unicode has 17 planes of 65,536 each totalling 1,114,112 characters
    - Only 10% are allocated so far, which gives us around 115,000 for b.

External sorting - working on disks/tapes

- Like Trees, there’s a need for sorting outside of main memory
  - B+ Trees were designed to solve this on disk
  - Sorting on tapes directly has a whole suite of options
  - Still relevant given the size of tapes: Sony’s are up to 185 TB per tape
    - IBM/Sony just did 330 TB/tape last month: https://goo.gl/n5CNds
  - Companies still do many backups to tape even today
- Primary issues include:
  - Only linear access to items on the tape with any speed (forward or backward)
  - Incredibly slow compared to RAM if done out of order
  - Can use multiple tapes in parallel if your hardware supports it
- We’re not going to spend too much time on this (actually, we’re done)

Sorting summary

- The ordering of data is a huge research field
- Small advances can save companies huge money
- Picking the right algorithm to use needs to understand:
  - Data to sort
    - Primary consideration is input size
    - Pre-sorted, or almost sorted situations
  - Programming environment behaviors
  - Opportunity to do linear time sorting
- Most situations call for: Insertion sort, shellsort, mergesort, or quicksort
  - Be aware of opportunities for linear time sorting via bucket or radix sorts

Monday: Probably not doing graphs yet

- We have one kind of free day next week:
  Monday: ??? - linux tools, more git for group work, etc
  Wednesday: Midterm review
  Friday: Midterm #2 - Heaps, Hashing, and Sorting

If I’ve gone fast enough, we can watch hacker video for a while

https://www.youtube.com/watch?v=hqKafI7Amd8