CSE373: Data Structure & Algorithms

Lecture 22: Beyond Comparison Sorting

Aaron Bauer

Winter 2014

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures…

Simple algorithms: O(n^2)	**Fancier algorithms: O($n \log n$)**	**Comparison lower bound: $\Omega(n \log n)$**	**Specialized algorithms: O(n)**	**Handling huge data sets**
Insertion sort **Selection sort** **Shell sort** …	**Heap sort** **Merge sort** **Quick sort (avg)** …		**Bucket sort** **Radix sort**	**External sorting**

How???

- Change the model – assume more than "compare(a,b)"

BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and *K* (or any small range):
  - Create an array of size *K*
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

`count` array	
1	3
2	1
3	2
4	2
5	3

- Example:

  K=5

  input (5,1,3,4,3,2,1,1,5,4,5)

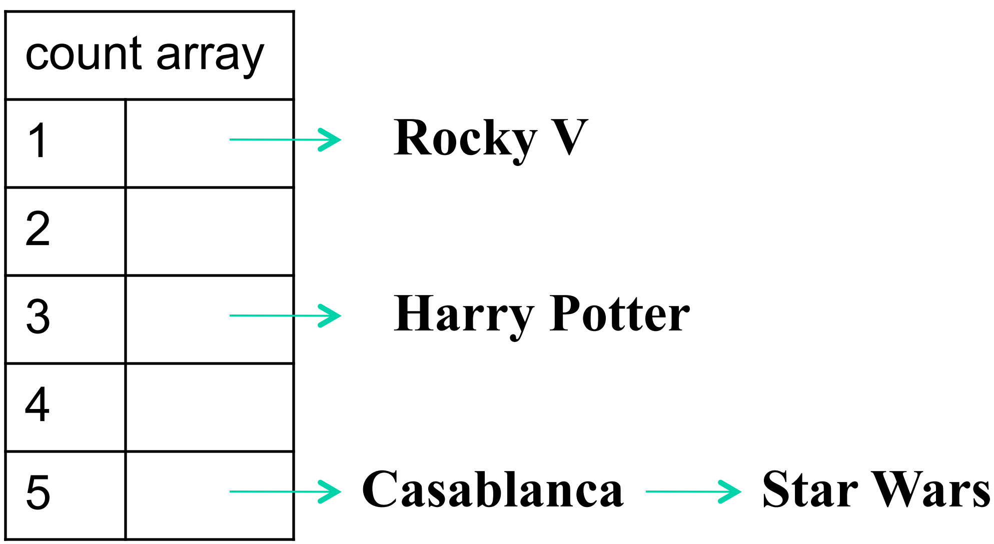
  output: 1,1,1,2,3,3,4,4,5,5,5

Analyzing Bucket Sort

- Overall: $O(n+K)$
  - Linear in n, but also linear in K
  - $\Omega(n \texttt{ log } n)$ lower bound does not apply because this is not a comparison sort
- Good when K is smaller (or not much larger) than n
  - We don't spend time doing comparisons of duplicates
- Bad when K is much larger than n
  - Wasted space; wasted time during linear $O(K)$ pass
- For data in addition to integer keys, use list at each bucket

Bucket Sort with Data

- Most real lists aren’t just keys; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, insert in *O*(1) (at beginning, or keep pointer to last element)



- Example: Movie ratings; scale 1-5;1=bad, 5=excellent

  Input=

  5: Casablanca

  3: Harry Potter movies

  5: Star Wars Original Trilogy

  1: Rocky V

- Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
- Easy to keep ‘stable’; Casablanca still before Star Wars

Radix sort

- Radix = "the base of a number system"
  - Examples will use 10 because we are used to that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128

- Idea:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least* significant digit
    - Keeping sort *stable*
  - Do one pass per digit
  - Invariant: After *k* passes (digits), the last *k* digits are sorted

- Aside: Origins go back to the 1890 U.S. census

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

Input: 478
537
9
721
3
38
143
67

First pass:

bucket sort by ones digit

Order now: 721
3
143
537
67
478
38
9

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Order was: 721
3
143
537
67
478
38
9

Second pass:

stable bucket sort by tens digit

Order now: 3
9
721
537
38
143
67
478

Example

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Radix = 10

0	1	2	3	4	5	6	7	8	9
3 9 38 67	143			478	537		721		

Order was:

3
9
721
537
38
143
67
478

Third pass:

stable bucket sort by 100s digit

Order now:

3
9
38
67
143
478
537
721

Analysis

Input size: n

Number of buckets = Radix: B

Number of passes = “Digits”: P

Work per pass is 1 bucket sort: $O(B+n)$

Total work is $O(P(B+n))$

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - Run-time proportional to: $15*(52 + n)$
  - This is less than n log n only if $n > 33{,}000$
  - Of course, cross-over point depends on constant factors of the implementations
    - And radix sort can have poor locality properties

Sorting massive data

- Need sorting algorithms that minimize disk/tape access time:
  - Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
  - Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access

- Mergesort is the basis of massive sorting

- Mergesort can leverage multiple disks

External Merge Sort

- Sort 900 MB using 100 MB RAM
  - Read 100 MB of data into memory
  - Sort using conventional method (e.g. quicksort)
  - Write sorted 100MB to temp file
  - Repeat until all data in sorted chunks (900/100 = 9 total)
- Read first 10 MB of each sorted chuck, merge into remaining 10MB
  - writing and reading as necessary
  - Single merge pass instead of *log n*
  - Additional pass helpful if data much larger than memory
- Parallelism and better hardware can improve performance
- Distribution sorts (similar to bucket sort) are also used

Last Slide on Sorting

- Simple $O(n^2)$ sorts can be fastest for small n
  - Selection sort, Insertion sort (latter linear for mostly-sorted)
  - Good for “below a cut-off” to help divide-and-conquer sorts
- $O(n$ **`log`** $n)$ sorts
  - Heap sort, in-place but not stable nor parallelizable
  - Merge sort, not in place but stable and works as external sort
  - Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies
- Ω (n **`log`** n) is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of possible key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!

What is a Programming Language?

- A set of symbols and associated tools that translate (if necessary) collections of symbols into instructions to a machine
  - Compiler, execution platform (e.g. Java Virtual Machine)
  - Designed by someone or some people
    - Can have flaws, poor decisions, mistakes
- Syntax
  - What combinations of symbols are allowed
- Semantics
  - What those combinations mean
- These can be defined in different ways for different languages
- There are a lot of languages
  - Wikipedia lists 675 excluding dialects of BASIC and esoteric languages

Before High-Level Languages

- Everything done machine code or an assembly language
  - Arithmetic operations (add, multiply, etc.)
  - Memory operations (storing, loading)
  - Control operations (jump, branch)
- Example: move 8-bit value into a register
  - 1101 is binary code for move followed by 3-bit register id
  - 1101000 01100001
  - B0 61
  - MOV AL, 61h ; Load AL with 97 decimal (61 hex)

A Criminally Brief History of Features

- First compiled high-level language: 1952 (Autocode)
- Math notation, subroutines, arrays: 1955 (Fortran)
- Recursion, higher-order functions, garbage collection: 1960 (LISP)
- Nested block structure, lexical scoping: 1960 (ALGOL)
- Object-orientated programming: 1967 (Simula)
- Generic programming: 1973 (ML)

Language timeline

- C: 1973
- C++: 1980
- MATLAB: 1984
- Objective-C: 1986
- Mathematic (Wolfram): 1988
- Python: 1991
- Ruby: 1993
- Java: 1995
- Javascript: 1995
- PHP: 1995
- C#: 2001
- Scala: 2003

What do we want from a Language?

- Performant
- Expressive
- Readable
- Portable
- Make dumb things difficult
- …

Type System

- Collection of rules to assign types to elements of the language
  - Values, variables, functions, etc.
- The goal is to reduce bugs
  - Logic errors, memory errors (maybe)
- Governed by type theory, an incredibly deep and complex topic

- The type safety of a language is the extent to which its type system prevents or discourages relevant type errors
  - Via type checking
- We'll cover the following questions:
  - When does the type system check?
  - What does the type system check?
  - What do we have to tell the type system?

When Does It Check?

- Static type-checking (check at compile-time)
  - Based on source code (program text)
  - If program passes, it's guaranteed to satisfy some type-safety properties on all possible inputs
  - Catches bugs early (program doesn't have to be run)
  - Possibly better run-time performance
    - Less (or no) checking to do while program runs
    - Compiler can optimize based on type
  - Inherently conservative
    - *if <complex test> then <do something> else <type error>*
  - Not all useful features can be statically checked
    - Many languages use both static and dynamic checking

When Does it Check?

- Dynamic type-checking (check at run-time)
  - Performed as the program is executing
  - Often “tag” objects with their type information
  - Look up type information when performing operations
  - Possibly faster development time
    - edit-compile-test-debug cycle
  - Fewer guarantees about program correctness

What Does it Check?

- Nominal type system (name-based type system)
  - Equivalence of types based on declared type names
  - Objects are only subtypes if explicitly declared so
  - Can be statically or dynamically checked
- Structural type system (property-based type system)
  - Equivalence of types based on structure/definition
  - An element A is compatible with an element B if for each feature in B's type, there's an identical feature in A's type
    - Not symmetric, subtyping handled similarly
- Duck typing
  - Type-checking only based on features actually used
  - Only generates run-time errors

How Much do we Have to Tell it?

- Type Inference
  - Automatically determining the type of an expression
  - Programmer can omit type annotations
    - Instead of (in C++)
      std::vector<int>::const_iterator itr = myvec.cbegin()
      use (in C++11)
      auto itr = myvec.cbegin()
  - Can make programming tasks easier
  - Only happens at compile-time
- Otherwise, types must be manifest (always written out)

What does it all mean?

- Most of these distinctions are not mutually exclusive
  – Languages that do static type-checking often have to do some dynamic type-checking as well
  – Some languages use a combination of nominal and duck typing
- Terminology useful shorthand for describing language characteristics
- The terms “strong” or “weak” typing are often applied
  – These lack any formal definition
  – Use more precise, informative descriptors instead
- Next lecture:
  – Overview of other important language attributes
  – Comparisons of common languages