Algorithm Design and Analysis

Integer models of computation and integer sorting

Roadmap for today

- Breaking out of the comparison model, the ***word-RAM***
- Learn about the ***Counting Sort*** algorithm
- Learn about the ***Radix Sort*** algorithm

Last lecture: Sorting can not be done faster than $\Omega(n \log n)$ ***in the comparison model***

Today: Sorting in $O(n)$ time ***for bounded integers in the word RAM model***

Formal model of computation

- We’re leaving the comparison model today. We want to take advantage of integer-ness for more performance!!

- ***Model (word-RAM)***:
  - We have unlimited constant-time addressable memory (“registers”)
  - Each register can store a w-bit integer (a “word”)
  - Reading/writing, arithmetic, logic, bitwise operations on a constant number of words takes constant time
  - With input size n, we need $w \geq \log n$.

- Default assumption is that w is large enough that all integers in the input to the problem fit in a single word

Implications of the word-RAM

- Adding two b-bit integers gives a $(b+1)$-bit integer.
- Multiplying two b-bit integers gives a $2b$-bit integer.
- A constant number of these is therefore okay since the result fits in a constant number of registers.

- What if we **multiply** $\underline{n\ w}$-bit integers? We get a $\underline{\Theta(nw)}$-bit answer! This ***does not fit*** in a single/constant number of registers!!
- Such an algorithm would therefore take ***more than*** $\Theta(n)$ time.

Real-life equivalent

```
int product = 1;
for (int i = 0; i < n; i++)
  product *= a[i];
```

- Too much addition/multiplication can quickly lead to ***overflow***

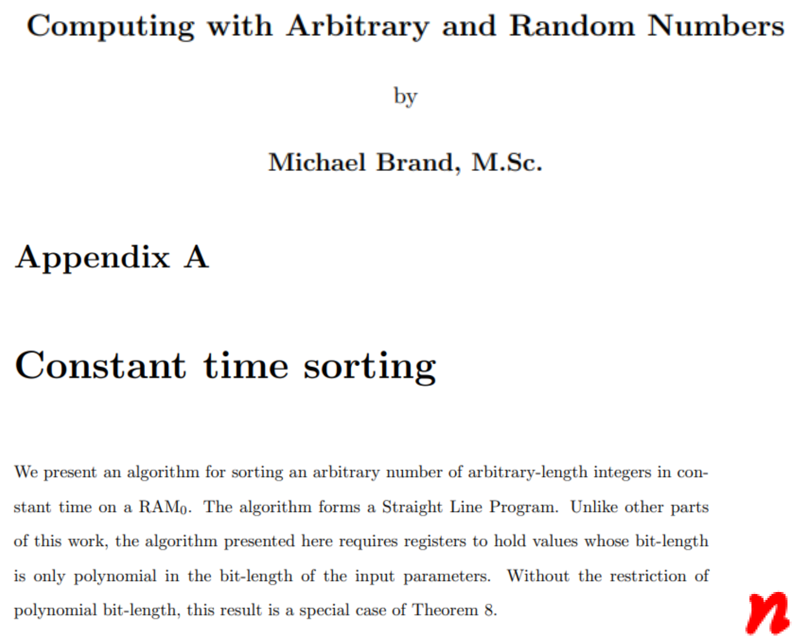
```
product = 1
for i in range(n):
 product *= a[i]
```

- Python will represent large integers for you, but multiplying them ***is not constant time***

- The word RAM model is just the <u>theoretical equivalent</u> of ***watch out for overflow***. Something you should already be thinking about when designing algorithms.

Do we *really* need to restrict *w*?

- Suppose we allow reading/writing/instructions on arbitrarily long integers
- This is usually called the unit-cost RAM (as opposed to the word-RAM)



Computing with Arbitrary and Random Numbers

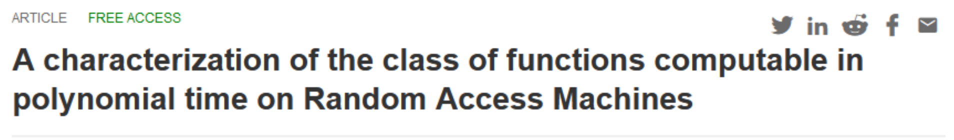
by

Michael Brand, M.Sc.

Appendix A

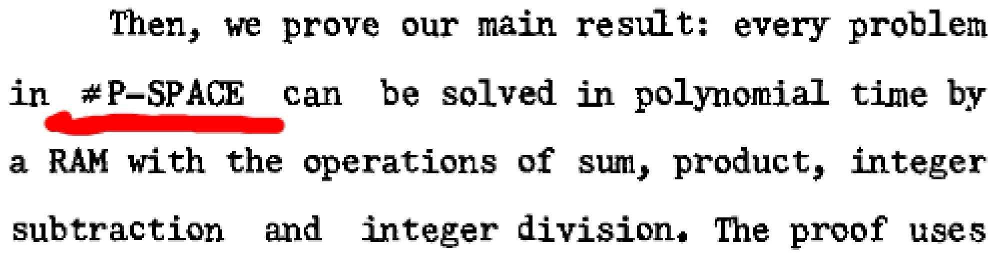
Constant time sorting

We present an algorithm for sorting an arbitrary number of arbitrary-length integers in constant time on a RAM$_0$. The algorithm forms a Straight Line Program. Unlike other parts of this work, the algorithm presented here requires registers to hold values whose bit-length is only polynomial in the bit-length of the input parameters. Without the restriction of polynomial bit-length, this result is a special case of Theorem 8.



ARTICLE FREE ACCESS

A characterization of the class of functions computable in polynomial time on Random Access Machines



Then, we prove our main result: every problem in #P-SPACE can be solved in polynomial time by a RAM with the operations of sum, product, integer subtraction and integer division. The proof uses

$n = 1000\cdots 1$ n^k $1\cdots 2\cdots 7$

Beating the comparison model

- As a warmup, consider the static searching problem

Problem (*Static search*) Given an array of elements $\underline{a_1, a_2, \dots, a_n}$, with arbitrary preprocessing allowed for free, determine the index of a query element x if it exists

- What is a lower bound for this problem in the comparison model?

$\Omega(\log n)$

information theoretic

lower bound.

Static searching in the word RAM

- Suppose the array of elements are ***integers*** and we are in the word RAM model of computation
- **Preprocessing**: Build a ***lookup table***: Store $\underline{i}$ at position $\underline{a_i}$
- **Query**: To search for x, just look in position x and see if its not empty

The power of the word RAM

- The fundamental limitation of the comparison model is the fact that we can only have binary (YES / NO) decisions!
- The word RAM bestows upon us the power of indirect addressing!
- A single instruction, e.g., lookup element i of an array of length n, can have n possible different outcomes!

Integer Sorting

Problem statement: Integer sorting

Problem (*Integer sorting*) Given an array of elements $a_1, a_2, \ldots, a_n$, each identified by a (not necessarily unique) **integer key** called $\underline{\boldsymbol{key(a_i)}}$, output an array containing a permutation $a_{\pi_1}, a_{\pi_2}, \ldots, a_{\pi_n}$ such that

$$key(a_{\pi_1}) \le key(a_{\pi_2}) \le \cdots \le key(a_{\pi_n})$$

- Duplicate keys are allowed!

eg: input [6, 5, 1, 8] $\pi = [3, 2, 1, 4]$
output [1, 5, 6, 8]

- Sorting is called ***stable*** if the relative order of duplicates is preserved
- Input contains arbitrary elements (not necessarily just integers) with ***integer keys***. Sorting must keep data + keys together.

Sorting unique very small integers

- We saw that we can beat the comparison model by taking advantage of indirect addressing (i.e., looking up in an array)
- ***Simpler problem***: Suppose keys are guaranteed to be *unique* integers in $\underline{\{1,2,\ldots,n\}}$

Algorithm:

- Create result array S of length n
- For each a_i, store $S[\underline{key(a_i)}] = a_i$
- S is the sorted answer!

$n \leq U$

Sorting unique small integers

- Now let's increase the size of the keys. Suppose the input elements all have unique keys in range $\underline{\{0,1, \ldots, u-1\}}$ (the parameter u is called the ***universe*** of keys)

Algorithm:

- Create result array S of length u $O(u)$
- For each $\underline{a_i}$, store $S[key(a_i)] = a_i$ $O(n)$
- Filter out the empty elements of S $O(u)$
- S is the sorted answer!

Time: $O(n+u)$

$n \leq u$

$= O(u)$

Sorting small integers

- Now let's remove the assumption that the keys are unique. Suppose the input elements have (not necessarily unique) keys in the range $\{0,1,\ldots,u-1\}$.

Algorithm (Counting Sort):

- Create a list for every possible key $\{0,1,\ldots,u-1\}$
- For each a_i, append a_i to list at index $key(a_i)$
- Concatenate all the lists together. Elements will be sorted!

Counting Sort

```
function CountingSort(a : array, key : element → int) {
    let L be an array of u empty lists
    for each element x in a do {
            L[key(x)].append(x) ←
    }
    let out be an empty list
    for each integer k in range(0, u) do {
            out.extend(L[k])                stable!
    }
    return out
}
```

Analysis of Counting Sort

Theorem: Counting Sort runs in $O(\underline{n+u})$ time.

```
function CountingSort(a : array, key : element → int) {
  let L be an array of u empty lists            O(u)
  for each element x in a do {                  O(n)
    L[key(x)].append(x)
  }
  let out = empty list
  for each integer k in range(0, u) do {        O(u+n)
    out.extend(L[k])
  }
  return out
}
```

Corollary: Counting Sort sorts keys in $\{0, 1, \ldots, O(n)\}$ in $\underline{O(n)}$ time!

Side quest: Tuple sorting

Problem (*Tuple sorting*) Given an array of elements $a_1, a_2, \dots, a_n$, each identified by a **tuple of keys** $(k_1, k_2, \dots, k_d)$, sort the array **lexicographically** by the tuple. That is, the array is sorted by k_1, with ties broken by k_2, and ties on that broken by k_3 and so on!

2024, Feb, 16		2023, Jan, 17
2024, Feb, 18		2023, Jan, 19
2023, Feb, 06		2023, Feb, 06
2024, Jan, 16		2023, Feb, 07
2023, Jan, 19		2023, Feb, 19
2023, Jan, 17	➡	2024, Jan, 16
2024, Feb, 06		2024, Jan, 18
2023, Feb, 07		2024, Jan, 19
2023, Feb, 19		2024, Feb, 06
2024, Feb, 15		2024, Feb, 15
2024, Jan, 19		2024, Feb, 16
2024, Jan, 18		2024, Feb, 18

Algorithms for tuple sorting

Algorithm (Comparison tuple sort): Just use your favourite comparison-sorting algorithm (MergeSort, HeapSort, QuickSort, etc.) and compare tuples lexicographically

- **Cost:** $\underline{O(d\,n\log n)}$ in the comparison model.
- Does not generalize well outside the comparison model

Top-down tuple sorting

Algorithm (Top-down tuple sort): Sort by the first tuple element, then recursively sort the ties on the second tuple element and so on…

2024, Feb, 16	2023, Feb, 06	2023, Jan, 19	2023, Jan, 17
2024, Feb, 18	2023, Jan, 19	2023, Jan, 17	2023, Jan, 19
2023, Feb, 06	2023, Jan, 17	2023, Feb, 06	2023, Feb, 06
2024, Jan, 16	2023, Feb, 07	2023, Feb, 07	2023, Feb, 07
2023, Jan, 19	2023, Feb, 19	2023, Feb, 19	2023, Feb, 19
2023, Jan, 17	2024, Feb, 16	2024, Jan, 16	2024, Jan, 16
2024, Feb, 06	2024, Feb, 18	2024, Jan, 19	2024, Jan, 18
2023, Feb, 07	2024, Jan, 16	2024, Jan, 18	2024, Jan, 19
2023, Feb, 19	2024, Feb, 06	2024, Feb, 16	2024, Feb, 06
2024, Feb, 15	2024, Feb, 15	2024, Feb, 18	2024, Feb, 15
2024, Jan, 19	2024, Jan, 19	2024, Feb, 06	2024, Feb, 16
2024, Jan, 18	2024, Jan, 18	2024, Feb, 15	2024, Feb, 18
	Sorted	Sorted	Sorted

Bottom-up tuple sorting

Algorithm (Bottom-up tuple sort): **Stable sort** by the **last** tuple element, then the **second last**, and so on, finally sorting by the **first** tuple element.

	2024, Feb, 16		2023, Feb, 06		2024, Jan, 16		2023, Jan, 17
	2024, Feb, 18		2024, Feb, 06		2023, Jan, 17		2023, Jan, 19
1	2023, Feb, 06		2023, Feb, 07		2024, Jan, 18		2023, Feb, 06
	2024, Jan, 16		2024, Feb, 15		2023, Jan, 19		2023, Feb, 07
	2023, Jan, 19		2024, Feb, 16		2024, Jan, 19		2023, Feb, 19
	2023, Jan, 17	→	2024, Jan, 16	→	2023, Feb, 06	→	2024, Jan, 16
2	2024, Feb, 06		2023, Jan, 17		2024, Feb, 06		2024, Jan, 18
	2023, Feb, 07		2024, Feb, 18		2023, Feb, 07		2024, Jan, 19
	2023, Feb, 19		2024, Jan, 18		2024, Feb, 15		2024, Feb, 06
	2024, Feb, 15		2023, Jan, 19		2024, Feb, 16		2024, Feb, 15
	2024, Jan, 19		2023, Feb, 19		2024, Feb, 18		2024, Feb, 16
	2024, Jan, 18		2024, Jan, 19		2023, Feb, 19		2024, Feb, 18
			Sorted		Sorted		Sorted

Sorting bigger integers

- Counting sort runs in linear ($O(n)$) time for $u = O(n)$
- We want to sort in linear time for bigger values of u
- **Cool idea:** *Use tuple sort* to sort integer keys?

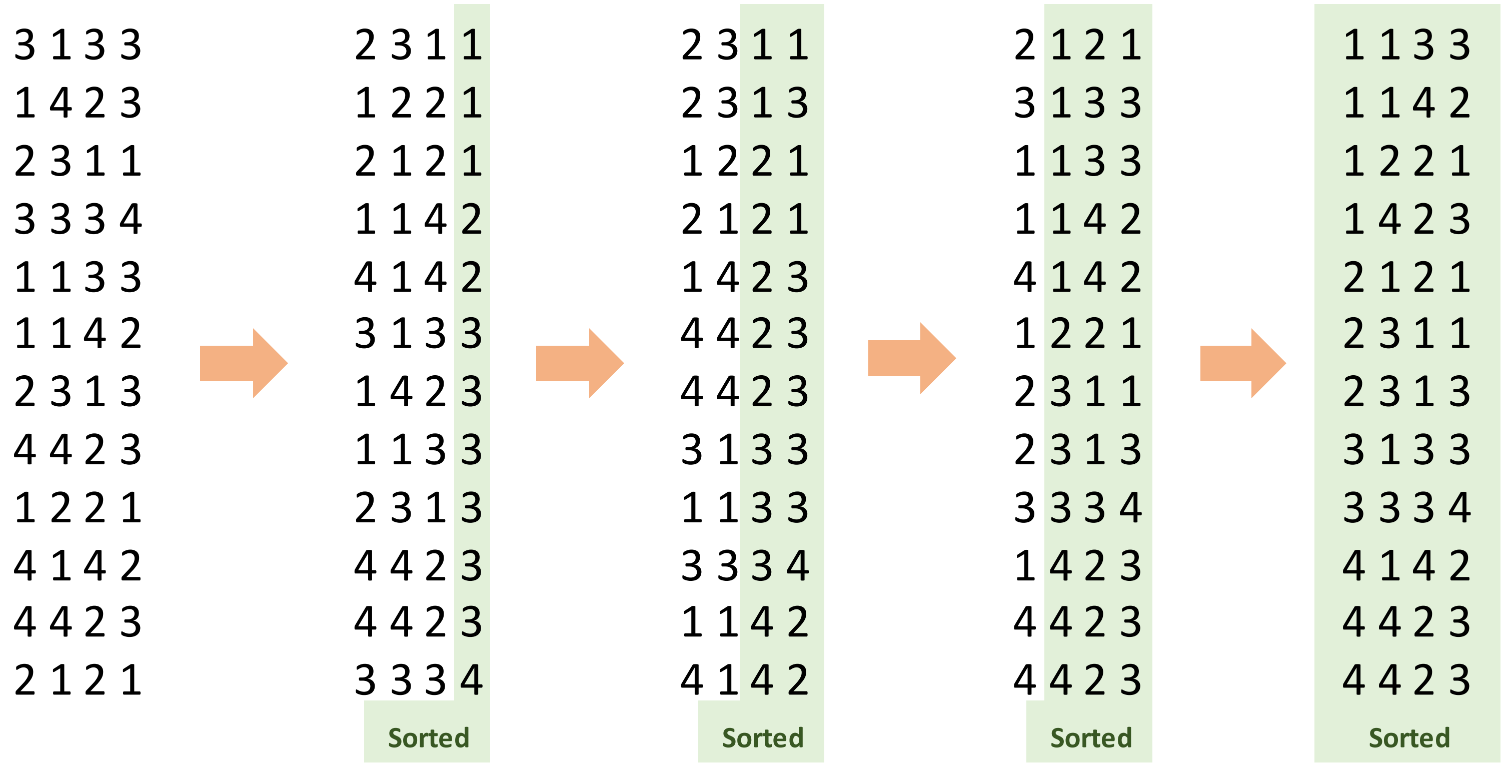
Question: Can we represent a big integer as a tuple of small integers such that tuple sorting them gives the right answer?

Answer: Just use their ***digits***! (Small integers so Counting Sort works)

any base

Bottom-up (LSD) Radix Sort

Algorithm (LSD Radix Sort): Counting sort by the **last** digit, then the **second last**, and so on, finally sorting by the **first** digit.



3 1 3 3	2 3 1 1	2 3 1 1	2 1 2 1	1 1 3 3
1 4 2 3	1 2 2 1	2 3 1 3	3 1 3 3	1 1 4 2
2 3 1 1	2 1 2 1	1 2 2 1	1 1 3 3	1 2 2 1
3 3 3 4	1 1 4 2	2 1 2 1	1 1 4 2	1 4 2 3
1 1 3 3	4 1 4 2	1 4 2 3	4 1 4 2	2 1 2 1
1 1 4 2	3 1 3 3	4 4 2 3	1 2 2 1	2 3 1 1
2 3 1 3	1 4 2 3	4 4 2 3	2 3 1 1	2 3 1 3
4 4 2 3	1 1 3 3	3 1 3 3	2 3 1 3	3 1 3 3
1 2 2 1	2 3 1 3	1 1 3 3	3 3 3 4	3 3 3 4
4 1 4 2	4 4 2 3	3 3 3 4	1 4 2 3	4 1 4 2
4 4 2 3	4 4 2 3	1 1 4 2	4 4 2 3	4 4 2 3
2 1 2 1	3 3 3 4	4 1 4 2	4 4 2 3	4 4 2 3
	Sorted	Sorted	Sorted	Sorted

Bottom-up (LSD) Radix Sort

```
function RadixSort(a : array, key : element → int) {
    let out = copy of a
    for each i in range(0, num_digits) do {
        out = CountingSort(out, key =
                    x → Digit(key(x), num_digits
                                        - i)
                    )
    }
    return out
}
```

Analysis of Radix Sort

Theorem: Radix Sort runs in $O((n+b)\underline{\log_b u)}$ time using base-b

counting sort : $O(n+b)$

iterations : # digits $= \log_b u$

Optimal choice of b?

- How do we optimize $O(\underline{(n+b)}\ \underline{\log_b u})$?
- Bigger base ⇒ fewer iterations (but also slower Counting Sort)

Optimal base: $b = n$

Running time: $O(n \log_n u)$

Radix Sort can sort big(ger) integers

Theorem: Radix Sort can sort keys in $\{0,1,\ldots,\underline{O(n^c)}\}$ in $O(n)$ time!

$\underline{O(n \cdot c)}$

$u = n^c$

$n \log_n u = O(n \log_n n^c)$

$= O(nc)$

Summary of Radix and Counting Sort

Given n input elements with integer keys in $\{0,1,\dots,u-1\}$,

- **Counting Sort** runs in $O(n+u)$ time.
  - This is linear time whenever $u = O(n)$, i.e., linear-sized keys
- **Radix Sort** runs in $O(n \log_n u)$ time.
  - This is linear time whenever $u = O(n^c)$, i.e., polynomial-sized keys!

Fun fact (Integer sorting is still an open problem): We don't know whether there exists an algorithm that can sort integers of any size in linear time. The best discovered algorithms take $O(n \operatorname{loglog} n)$ time (deterministic) or $O\left(n\sqrt{\operatorname{loglog} n}\right)$ expected time. No known lower bound proves that linear-time integer sorting is impossible, but we don't know!