Data Structures and Algorithms

Sorting

Bin Sorts

الدكتور
اثير العاني

Key Points

- **Quicksort**
  - **Use for good overall performance where time is not a constraint**
- **Heap Sort**
  - **Slower than quick sort, but guaranteed $O(n \log n)$**
  - **Use for real-time systems where time is critical**
- **Functions as data types**
  - **Argument of a function can be a function**
  - **Enables flexible general purpose classes**
  - **Enables table driven code**

Sorting

- **We now know several sorting algorithms**
  - **Insertion** $O(n^2)$
  - **Bubble** $O(n^2)$
  - **Heap** $O(n \log n)$ *Guaranteed*
  - **Quick** $O(n \log n)$ *Most of the time!*
- **Can we do any better?**

Sorting - Better than $O(n \log n)$ *?*

- **If all we know about the keys is an ordering rule**
  - **No!**
- **However,**
  - ***If we can compute an address from the key***
    (in constant time) **then**
    bin sort algorithms can provide better performance

Sorting - Bin Sort

- **Assume**
  - **All the keys lie in a small, fixed range**
    - ***eg***
      - **integers 0-99**
      - **characters ‘A’-’z’, ‘0’-’9’**
  - **There is at most one item with each value of the key**
- **Bin sort**
  - ✶ **Allocate a bin for each value of the key**
    - **Usually an entry in an array**
  - ✷ **For each item,**
    - **Extract the key**
    - **Compute it’s bin number**
    - **Place it in the bin**
  - ✹ **Finished!**

Sorting - Bin Sort: Analysis

- All the keys lie in a small, fixed range
  - There are m possible key values
- There is at most one item with each value of the key

- Bin sort
  - ✶ Allocate a bin for each value of the key $O(m)$
    - Usually an entry in an array
  - ✷ For each item, *n times*
    - Extract the key $O(1)$
    - Compute it's bin number $O(1)$
    - Place it in the bin $O(1)$ x n 🡐 $O(n)$
  - ✹ Finished! $O(n) + O(m) = O(n+m) = \boxed{O(n)}$ if $n >> m$

Key condition

Sorting - Bin Sort: Caveat

- **Key Range**
  - **All the keys lie in a small, fixed range**
    - **There are m possible key values**
  - **If this condition is not met, *eg* $m >> n$, then bin sort is $O(m)$**
- **Example**
  - **Key is a 32-bit integer, $m = 2^{32}$**
  - **Clearly, this isn't a good way to sort a few thousand integers**
  - **Also, *we may not have enough space for bins!***
- **Bin sort trades space for speed!**
  - **There's no free lunch!**

Sorting - Bin Sort with duplicates

Relax?

- ~~There is at most one item with each value of the key~~

- Bin sort
  - Allocate a bin for each value of the key $O(m)$
    - Usually an entry in an array
    - Array of list heads
  - For each item, n *times*
    - Extract the key $O(1)$
    - Compute it's bin number $O(1)$
    - Add it to a list $O(1) \times n \leftarrow O(n)$
    - Join the lists $O(m)$
    - Finished! $O(n) + O(m) = O(n+m) = \boxed{O(n)}$ *if* $n >> m$

Sorting - Generalised Bin Sort

- **Radix sort**
  - **Bin sort in phases**
  - **Example** `36 9 0 25 1 49 64 16 81 4`

  - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

Sorting - Generalised Bin Sort

- **Radix sort** - **Bin sort in phases**
  - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

  - **Phase 2 - Sort by most significant digit**

0	1	2	3	4	5	6	7	8	9
0									

Sorting - Generalised Bin Sort

- **Radix sort** - **Bin sort in phases**
  - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

  - **Phase 2 - Sort by most significant digit**

0	1	2	3	4	5	6	7	8	9
0									
1									

Be careful to add after anything in the bin already!

Sorting - Generalised Bin Sort

- **Radix sort** - **Bin sort in phases**
  - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

  - **Phase 2 - Sort by most significant digit**

0	1	2	3	4	5	6	7	8	9
0								81	
1									

Sorting - Generalised Bin Sort

- **Radix sort** - Bin sort in phases
  - Phase 1 - Sort by least significant digit

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

  - Phase 2 - Sort by most significant digit

0	1	2	3	4	5	6	7	8	9
0						64		81	
1									

Sorting - Generalised Bin Sort

- **Radix sort - Bin sort in phases**
  - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

  - **Phase 2 - Sort by most significant digit**

0	1	2	3	4	5	6	7	8	9
0						64		81	
1									
4									

Sorting - Generalised Bin Sort

- **Radix sort** - **Bin sort in phases**
  - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

  - **Phase 2 - Sort by most significant digit**

0	1	2	3	4	5	6	7	8	9
0	16	25	36	49		64		81	
1									
4									
9									

Note that the 0 bin had to be quite large!

Sorting - Generalised Bin Sort

- **Radix sort** - **Bin sort in phases**
    - **Phase 1 - Sort by least significant digit**

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			49

-
    - **Phase 2 - Sort by most significant digit**

0	1	2	3	4	5	6	7	8	9
0	16	25	36	49		64		81	
1									
4									
9									

How much space is needed in each phase?

n **items**

m **bins**

Sorting - Generalised Bin Sort

- **Radix sort - Analysis**
  - **Phase 1 - Sort by least significant digit**
    - **Create m bins** $O(m)$
    - **Allocate n items** $O(n)$
  - **Phase 2**
    - **Create m bins** $O(m)$
    - **Allocate n items** $O(n)$
  - **Final**
    - **Link m bins** $O(m)$
  - **All steps in sequence, so add**
  - **Total** $O(3m+2n) \Rightarrow O(m+n) \Rightarrow O(n)$ *for* $m \ll n$

Sorting - Radix Sort - Analysis

- **Radix sort - General**
  - **Base (or radix) in each phase can be anything suitable**
  - **Integers**
    - **Base 10, 16, 100, …**
  - ***Bases don't have to be the same***

```
struct date {
    int day;   /* 1 .. 31 */
    int month; /* 1 .. 12 */
    int year;  /* 0 .. 99 */
    }
```

Phase 1 - s_1=31 bins
Phase 2 - s_2=12 bins
Phase 3 - s_3=100 bins

- **Still $O(n)$ if $n >> s_i$ for all i**

Radix Sort - Analysis

- **Generalised Radix Sort Algorithm**

For each of k radices

`radixsort( A, n ) {`
`  for(i=0;i<k;i++) {`
`    for(j=0;j<s[i];j++) bin[j] = EMPTY;`
	$O(s_i)$

`    for(j=0;j<n;j++) {`
`      move A[i]`
`      to the end of bin[A[i]->fi]`
`      }`
	$O(n)$

`    for(j=0;j<s[i];j++)`
`      concat bin[j] onto the end of A;`
`    }`
`}`
	$O(s_i)$

Radix Sort - Analysis

- **Generalised Radix Sort Algorithm**

`radixsort( A, n ) {`
`for(i=0;i<k;i++) {`
`for(j=0;j<s[i];j++) bin[j] = EMPTY;`
	$O(s_i)$

`for(j=0;j`
`move A`
`to the end of bin[A[i]->fi]`
`}`
	$O(n)$

`for(j=0;j<s[i];j++)`
`concat bin[j] onto the end of A;`
`}`
`}`
	$O(s_i)$

Clear the s_i bins for the i^{th} radix

Radix Sort - Analysis

- **Generalised Radix Sort Algorithm**

`radixsort( A, n ) {`
`   for(i=0;i<k;i++) {`
`      for(j=0;j<s[i];j++) bin[j] = EMPTY;`
	$O(s_i)$

`      for(j=0;j<n;j++) {`
`         move A[i]`
`         to the end of bin[A[i]->fi]`
`         }`
	$O(n)$

`      for(j=0;j`
`         concat`
`      }`
`   }`
	$O(s_i)$

Move element `A[i]`
to the end of the bin addressed
by the *i*

by the i^{th} field of `A[i]`

Radix Sort - Analysis

- **Generalised Radix Sort Algorithm**

```
radixsort( A, n ) {
   for(i=0;i<k;i++) {
      for(j=0;j<s[i];j++) bin[j] = EMPTY;
```

$O(s_i)$

```
      for(j=0;j<n;j++) {
         move A[i]
         to the en
         }
```

$O(n)$

Concatenate s_i bins into one list again

```
      for(j=0;j<s[i];j++)
         concat bin[j] onto the end of A;
      }
   }
```

$O(s_i)$

Radix Sort - Analysis

- **Total**
  - k **iterations,** $2s_i + n$ **for each one**

$$\sum_{i=1}^{k} O(s_i + n) = O(kn + \sum_{i=1}^{k} s_i)$$

$$= O(n + \sum_{i=1}^{k} s_i)$$

  - **As long as k is constant**
  - ***In general,* if the keys are in $(0, b^k\text{-}1)$**
    - **Keys are k-digit base-b numbers**

    ← $s_i = b$ for all k

← Complexity $O(n+kb) = O(n)$

Radix Sort - Analysis

? ***Any*** **set of keys can be mapped to** $(0, b^k-1)$
   ! **So we can always obtain** $O(n)$ **sorting?**
      • **If** k **is constant,** ***yes***

Radix Sort - Analysis

- But, if k is allowed to increase with n

  eg it takes $\log_b n$ base-b digits to represent n
- *so* we have
- $$k = \log n, \quad s_i = 2 \ (say)$$

➔ $$\sum_{i=1}^{\log n} O(2 + n) = O(n \log n + \sum_{i=1}^{\log n} 2)$$

$$= O(n \log n + 2 \log n)$$

$$= O(n \log n)$$

➔ **Radix sort is no better than quicksort**

Radix Sort - Analysis

- **Radix sort is no better than quicksort**
  - **Another way of looking at this:**
    - **We can keep k constant as n increases** ***if we allow duplicate keys***
      - **keys are in $(0, b^k)$, $b^k < n$**
  - ***but* if the keys must be unique, then k must increase with n**
- ***For $O(n)$ performance, the keys must lie in a restricted range***

Radix Sort - Realities

- **Radix sort uses a lot of memory**
  - $n\ s_i$ **locations for each phase**
  - **In practice, this makes it difficult to achieve** $O(n)$ **performance**
  - **Cost of memory management outweighs benefits**

Key Points

- **Bin Sorts**
  - ***If* a function exists which can convert the key to an address (*ie* a small integer)**

    ***and* the number of addresses (= number of bins) is not too large**

    ***then* we can obtain $O(n)$ sorting**

    ***… but remember it's actually* $O(n + m)$**
  - **Number of bins, m, must be *constant and small***