Data Structures and Algorithms

Semester 2, Sorting 6

Grouping based integer sorting

Counting sort
Bucket sort
Radix sort

Reading: Goodrich Chapter 11
Donald Knuth: The Art of Computer Programming, Volume 3: Sorting and Searching

Sorting not based on swapping and comparison

•The best we can do with sorting by comparison and swapping is O(nlogn)

•If you are sorting objects that can only take on a limited number of values, it can be better to use sorting algorithms that are not based on swapping

1, 1, 3, 1, 2, 2, 3, 3, 1, 2, 2, 3, 1, 1, 2, 2, 3, 3, 1, 1.

•Essentially we can exploit the fact that the variable we are sorting is discrete and has a finite range to use extra memory to improve speed.

Distribution Based Sorting

- **Counting sort:**
  Count the occurrences of each key, then construct sorted list from the histogram
- **Bucket sort:**
  Store the occurrences of elements in a value range (buckets), then construct the list from buckets
- **Radix sort:**
  grouping keys by the individual digits which share the same significant position and value

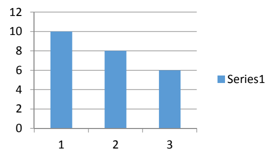
Counting Sort (informal)

32 bit integers have over 4 billion different values! That is a big histogram (which needs initialising)

Scan through array and count the occurrences of each number (form the histogram)

Linear time

1, 1, 3, 1, 2, 1, 2, 1, 3, 3, 1, 2, 2, 3, 1, 1, 2, 2, 3, 3, 1, 1, 2, 2.



Reconstruct sorted array from histogram

Linear time

the 1's start here

the 1's end here

hence the 2's start here

2's end here

3's start here

End here

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3

Counting Sort

Task: CountingSort: Sorts Array *T* into ascending order,
Where *T* are integers in range 1…k

```
begin countingSort(T[1…n ]
    Initialise counts[1…k] to zero
    for i:=1 to n
        counts[T[ i ]]:=counts[T[ i ]]+1
    cumulativeSum:=counts[1]
    j:=1
    T[1]:=j
    for i:=2 to n
        if i< cumulativeSum
                T[i]:=T[i-1]
        else
                j:=j+1
                T[i]:=j
                cumulativeSum:= cumulativeSum+counts[j]
```

Form histogram O(n)

copy back into T

This is Order(k+n)

*if k<nlog(n) then it **may** be better for sorting*

Counting Sort Arrays of Integers

- **Basis: Grouping**
- **Time Complexity:**
  O(k+n) comparisons, where k is the number of distinct values
- **Space Complexity:**
  - Requires and extra O(k) memory for the histogram
- **Stability**:
  - Counting sort is stable

Counting Sort Objects

- The histogram algorithm described only works for integers, where we do not need to store references to each object
- If we can use it for objects if we can associate each object with a unique **key.**
- The number of possible keys is **k**. A fixed size hash table (or hash map) with chaining is a simple way of implementing counting sort for objects (i.e. an array of linked lists)

Counting Sort Objects

Objects **A, A, C, A, B, A, B, A, C, C, A, B, B, C, A, A, B, B, C, C, A, A, B, B.**

keys **1, 1, 3, 1, 2, 1, 2, 1, 3, 3, 1, 2, 2, 3, 1, 1, 2, 2, 3, 3, 1, 1, 2, 2.**

Form hash table

1: A,A,A,A,A,A,A,A,A,A
2: B,B,B,B,B,B,B,B
3: C,C,C,C,C,C

Reconstruct sorted array by iterating over hash table in key order

Linear time

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
A	A	A	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	C	C	C	C	C	C

Counting Sort for Objects

Task: CountingSort: Sorts Array *T* into ascending order,
Where *T* are objects with keys in range 1…k
begin countingSort(T[*1…n*]
    Initialise countsMap to size *k*
    for *i*:=1 **to** n
        key:=T[*i*].getUniqueKey()
        countsMap.add(T[*i*], key)
    Iterator it:=countsTable.iterator()
    pos:=1
    While it.hasNext()
        T[*pos*++]:=*it.next()*

Form hash map where the element is stored in position k

Iterate over table in ***key order***

This is Order(k+n)

Counting Sort for Objects alternative algorithm

Task: Counting Sort*: Sorts Array T into ascending order,*
Where T are objects with keys in range 1…k
begin countingSort(T[*1…n*]
    Initialise buckets[k] to k empty lists
    //Insert elements into buckets by the unique key value
    for *i*:=1 **to** n
        key:=T[*i*].getUniqueKey()
        buckets[key].add(T[*i*])
    //copy each bucket back into T
    size:=1
    for *i* :=1 **to** k
        copy(T, buckets[*i*],size,size+buckets[i].size)
        size+=buckets[*i*].size

O(k) operation

If keys are the same, the objects are the same

Constant time add to a list

O(n) operation

This is Order(k+n)

Counting Sort Arrays of Objects

- **Basis: Grouping**
- **Time Complexity:**
  O(k+n) comparisons, where k is the number of distinct **keys**
- **Space Complexity:**
  - Requires and extra O(n) memory for the array of linked lists
- **Stability**:
  - Counting sort is stable if the hash table iterator is defined correctly

Problems with Counting Sort

- **Need to know *k* in advance, or at least bound it so that *k* is much smaller than *nlog(n)*.** Otherwise you will need to sort the keys!
- The bound on *k* is often much bigger than ***nlog(n)***

32 bit integers have over 4,294,967,296
For n=150,000,000, nlog(n)< 4,294,967,296

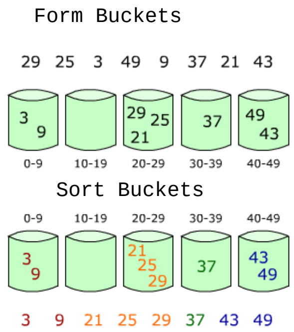
- Sorting on real values essentially requires heavy rounding
- The memory requirement can be high
- Bucket sort is variant of count sort that groups keys into bins, then sorts each bin

Bucket Sort (informal)

•Bucket sort, or bin sort works by partitioning an array into a number of buckets.

•Each bucket is then sorted individually

1. Set up array of buckets
2. Scan array, putting each number in a bucket
3. Scan buckets, sorting each
4. Copy back to array



Complexity depends on number of bins (m) and total number of elements. Average case O(n+m).

Bucket Sort

Task: BucketSort: Sorts Array T into ascending order, using m buckets

begin bucketSort(T[*1…n*], m)

Initialise buckets[m] to m empty lists

//Insert elements into buckets by some key value, so that the largest element in //bucket i is smaller than the smallest element in bucket i+1

for *i*:=1 **to** n

key=T[*i*].getKey()

buckets[key].add(T[*i*])

//sort each bucket

for *i*:=1 **to** m

countSort(buckets[*i*])

//copy each bucket back into T

size:=1

for *i*:=1 **to** m

copy(T, buckets[*i*],size,size+buckets[i].size)

size+=buckets[*i*].size

The only difference to count sort is that now if keys are the same, the objects are not necessarily equal

This means we have to sort each bucket before copying back

Bucket Sort: Forming Buckets

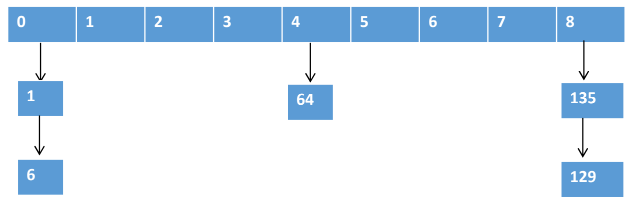
- Clearly, the efficiency of the algorithm depends on the distribution over the buckets (e.g. if everything is in one bucket, then the algorithm degenerates to the sort routine)
- Buckets are formed from a fixed number of leading terms of the elements.
- Usually, this is formed from the first *m* bits. So, for example, if we have 8 bit integers and 8 buckets, we could use the first four bits to determine the bucket

Integer	Binary	Bucket
6	00000110	0000=Bucket 0
64	01000000	0100=Bucket 4
129	10000001	1000=Bucket 8
135	10000111	1000=Bucket 8
1	00000001	0000=Bucket 0

Bucket Sort: Filling Buckets

- Each bucket is a linked list

Integer	Binary	Bucket
6	00000110	0000=Bucket 0
64	01000000	0100=Bucket 4
129	10000001	1000=Bucket 8
135	10000111	1000=Bucket 8
1	00000001	0000=Bucket 0



Bucket Sort: Sorting

- Any sorting algorithm can be used (counting sort is often chosen).
- If the values are uniformly distributed over the range, then bucket sort is O(n+m)
- Bucket sort can be used with any type of object (Strings, etc) where a key can be defined.
- Informally, it can be useful when there are too many values for counting sort.

Bucket Sort

•**Basis: Grouping**

•**Time Complexity:**

If counting sort is used, O(m+n) comparisons, where m is the number of buckets

•**Space Complexity:**

• Requires and extra O(n+m) memory for the bucket list and if counting sort is used O(mk) for the histograms (where k is the number of distinct values in each bucket)

•**Stability**:

•Bucket sort is stable, if the sub routine sorting is stable

Order Based Sorting

- Bucket sort relies on the fact we can split some data into sub units which can be assessed independently.
- So to compare decimal integers, we can split the number into separate values within the range of the radix
- So for example, with base 10 integers, on a range $\mathbf{0 \ldots 10^{d}}$

We can write any integer as

$$\boldsymbol{x = x_1 \times 10^{d-1} + x_2 \times 10^{d-2} + x_3 \times 10^{d-3} \ldots x_{d-1} \times 10^1 + x_d \times 10^0}$$

e.g $\mathbf{754553 = 7 \times 10^5 + 5 \times 10^4 + 4 \times 10^3 + 5 \times 10^2 + 5 \times 10^1 + 3 \times 10^0}$

Bucket Sort

Group by largest significant term into buckets

Sort each bucket

Radix Sort

Radix based sorting first sorts the elements in position *d*, then *d-1* etc

e.g $\mathbf{754553 = 7 \times 10^5 + 5 \times 10^4 + 4 \times 10^3 + 5 \times 10^2 + 5 \times 10^1 + 3 \times 10^0}$

$$\boldsymbol{x = x_1 \times 10^{d-1} + x_2 \times 10^{d-2} + x_3 \times 10^{d-3} \ldots x_{d-1} \times 10^1 + x_d \times 10^0}$$

■ ■ ■

Stably sort all elements by this digit

Then sort them again by this digit

Finally, sort them again by this digit

As if by magic, the array is now sorted!

Radix Sort example

Given integer data 11,123, 461,78,2457,7,3

Stably sort by the least significant digit 0011, 0461, 0123,0003, 2457,0007, 0078

Stable sort by the second least significant digit 0003,0007, 0011,0123,2457,0461, 0078

Stable sort by the third least significant 0003, 0007, 0011, 0078, 0123, 2457,0461,

Stable sort by the fourth least significant 0003, 0007, 0011, 0078, 0123, 0461, 2457

Radix Sort

Task: Radix Sort: Sorts Array T into ascending order,
Where T can be written as d-tuples, for a given radix
begin radixSort(T[*1…n*], d)
    for *i*:=d **to** 1
    //Usually count sort or bucket sort are used
        countSort(T,*i*,radix)

Radix Sort Example

11,123, 461,78,457,7,3,19,48,75,234,99,765,334,100,67,397,854

Need three rounds of sorting. Pad with zeros

011,123,461,078,457,007,003,019,048, 075,234,099,765,334,100,067,397,854

Round 1

0	100						
1	011	461					
2							
3	123	003					
4	234	334	854				
5	075	765					
6							
7	457	007	067	397			
8	078	048					
9	019	099					

Radix Sort Example

Map back into array

100,011,461,123,003,234,334,854,075, 765,457,007,067,397,078,048,019,099

Stably sort by second significant figure

Round 2

0	100	003	007				
1	011	019					
2	123						
3	234	334					
4	048						
5	854	457					
6	461	765	067				
7	075	078					
8							
9	397	099					

Radix Sort Example

Map back into array

100,003,007,011,019,123,234,334,048,
854, 457,461,765,067,075,078,397,099

Stably sort by third significant figure

Round 3

0	003	007	011	019	048	067	075	078	099
1	100	123							
2	234								
3	334	397							
4	457	461							
7	765								
8	854								
9									

Map back into array

003,007,011,019,048,067,075,078,099,
100,123,234,334,397,457,461,765,854

SORTED!

Radix Sort

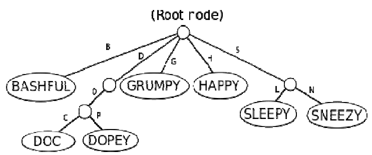
- **Basis: Grouping**
- **Time Complexity:**
  If counting sort is used, O(l*(k+n)) comparisons, where k is the radix (number of integers) and l is the word length (number of significant digits)

- **Space Complexity:**
  - Requires and extra O(n) memory for the histograms

  Stability:
  - Radix sort is stable

Radix Sort Extensions

- The examples we have looked at perform **Least Significant Digit** radix sort. Most significant digit radix sort is an alternative that can be implemented with **Tries**(example from wiki)



- We used a radix of 10 for clarity. It can be anything, and the best choice is data dependent
- There are efficient parallel radix sorting algorithms

https://arxiv.org/ftp/arxiv/papers/1511/1511.03404.pdf

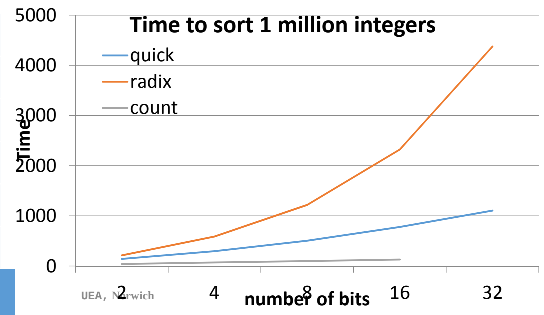
Experimental Comparison

Objective: What sorting algorithm should I use if there is a restriction on the number of values we can observe, assuming a random distribution over values

Method: Compare integer quick sort to count sort and radix sort for 1 million 32 bit, 16 bit, 8 bit, 3 bit and 2 bit integers. Average over 10 runs

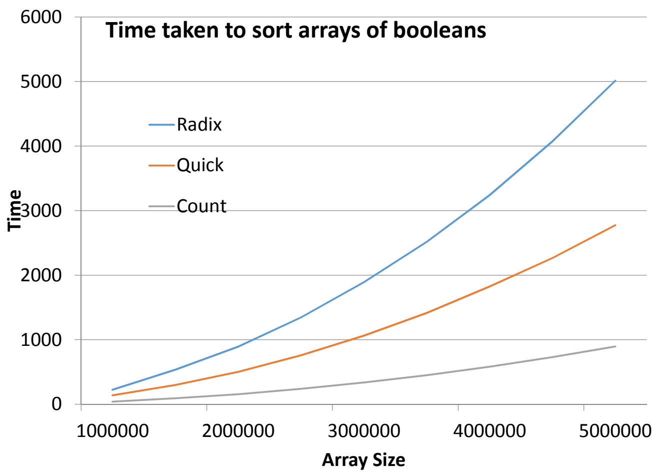
Bits	radix	quick	count
2	213	142	42
4	588	297	72
8	1220	506	101
16	2324	778	131
32	4375	1104	↑

Ran out of memory



Experimental Comparison

	Radix	Quick	Count
1000000	224	139	41
1500000	537	300	92
2000000	893	503	156
2500000	1345	759	239
3000000	1892	1062	337
3500000	2516	1414	451
4000000	3238	1824	581
4500000	4072	2265	730
5000000	5013	2775	895



Be careful inferring too much from this. The count sort does not actually shift the items, and as such would be slower with objects rather than integers

After Sorting Lecture 6 you should be able to …

1. **Describe count sort, bucket sort and radix sort both informally and in formal pseudo code**
2. **Know what their complexity is**
3. **Count sort, bucket sort and radix sort and array of integers**