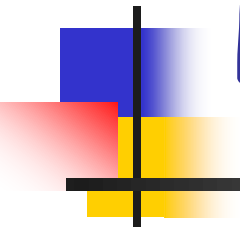


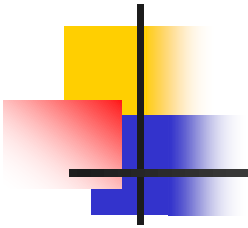
Bucket Sort and Radix Sort

Time complexity of Sorting

- the best ones, so far:
  - Heap sort and Merge sort: $O(n \log n)$
  - Quick sort (best one in practice): $O(n \log n)$ on average, $O(n^2)$ worst case
- Can we do better than $O(n \log n)$?
  - No.
  - It can be proven that any comparison-based sorting algorithm will need to carry out at least $O(n \log n)$ operations

Restrictions on the problem

- Suppose the values in the list to be sorted can repeat but the values have a limit (e.g., values are digits from 0 to 9)
- Sorting, in this case, appears easier
- Is it possible to come up with an algorithm better than $O(n \log n)$?
  - Yes
  - Strategy will not involve comparisons

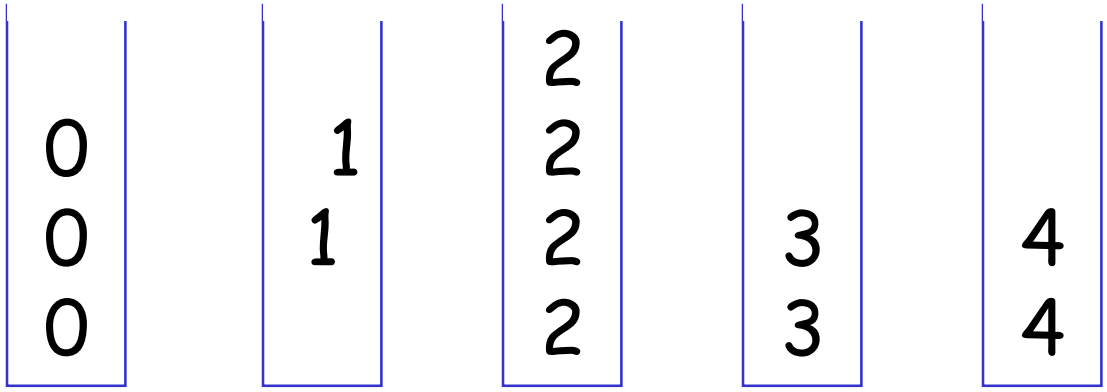


Bucket sort

- Idea: suppose the values are in the range 0..m-1; start with m empty *buckets* numbered 0 to m-1, scan the list and place element s[i] in bucket s[i], and then output the buckets in order
- Will need an array of buckets, and the values in the list to be sorted will be the indexes to the buckets
  - No comparisons will be necessary

Example

4	2	1	2	0	3	2	1	4	0	2	3	0

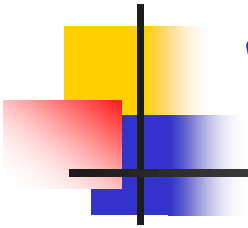


0	0	0	1	1	2	2	2	2	3	3	4	4

Bucket sort algorithm

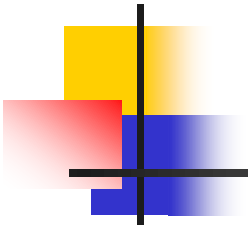
Algorithm BucketSort(S)
(values in S are between 0 and m-1)

```
for j ← 0 to m-1 do          // initialize m buckets
   b[j] ← 0
for i ← 0 to n-1 do          // place elements in their
   b[S[i]] ← b[S[i]] + 1     // appropriate buckets
i ← 0
for j ← 0 to m-1 do          // place elements in buckets
   for r ← 1 to b[j] do      // back in S
         S[i] ← j
         i ← i + 1
```



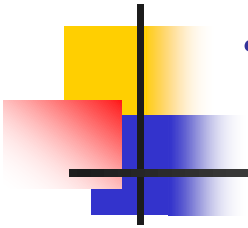
Values versus entries

- If we were sorting values, each bucket is just a counter that we increment whenever a value matching the bucket's number is encountered
- If we were sorting entries according to keys, then each bucket is a queue
  - Entries are enqueued into a matching bucket
  - Entries will be dequeued back into the array after the scan



Bucket sort algorithm

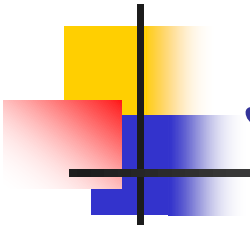
```
Algorithm BucketSort( S )
( S is an array of entries whose keys are between 0..m-1 )

for j ← 0 to m-1 do                    // initialize m buckets
   initialize queue b[j]
for i ← 0 to n-1 do                    // place in buckets
   b[S[i].getKey()].enqueue( S[i] );
i ← 0
for j ← 0 to m-1 do                    // place elements in
   while not b[j].isEmpty() do         // buckets back in S
         S[i] ← b[j].dequeue()
         i ← i + 1
```



Time complexity

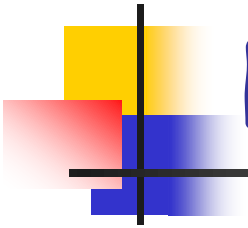
- Bucket initialization: $O(m)$
- From array to buckets: $O(n)$
- From buckets to array: $O(n)$
  - Even though this stage is a nested loop, notice that all we do is dequeue from each bucket until they are all empty -> n dequeue operations in all
- Since m will likely be small compared to n, Bucket sort is $O(n)$
  - Strictly speaking, time complexity is $O (n + m)$

Sorting integers



- Can we perform bucket sort on any array of (non-negative) integers?
  - Yes, but note that the number of buckets will depend on the maximum integer value
- If you are sorting 1000 integers and the maximum value is 999999, you will need 1 million buckets!
  - Time complexity is not really O(n) because m is much > than n. Actual time complexity is O(m)
- Can we do better?

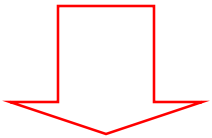
Radix sort



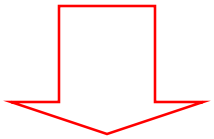
- Idea: repeatedly sort by digit—perform multiple bucket sorts on S starting with the rightmost digit
- If maximum value is 999999, only ten buckets (not 1 million) will be necessary
- Use this strategy when the keys are integers, and there is a reasonable limit on their values
  - Number of passes (bucket sort stages) will depend on the number of digits in the maximum value

Example: first pass

12	58	37	64	52	36	99	63	18	9	20	88	47



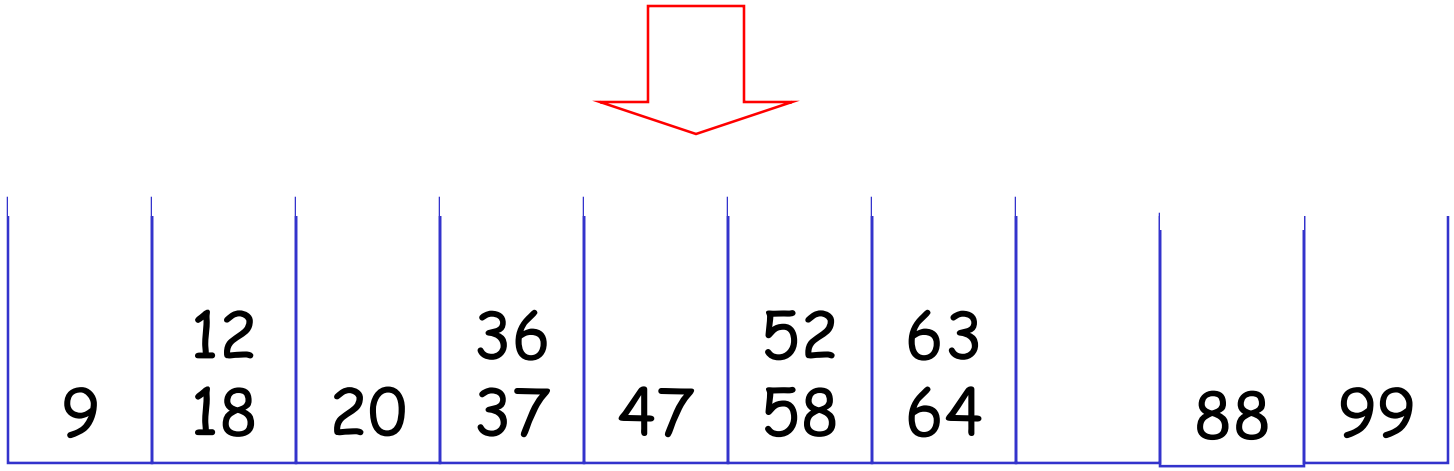
20		12 52	63	64		36	37 47	58 18 88	9 99	



20	12	52	63	64	36	37	47	58	18	88	9	99

Example: second pass

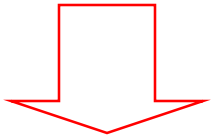
20	12	52	63	64	36	37	47	58	18	88	9	99



9	12	18	20	36	37	47	52	58	63	64	88	99

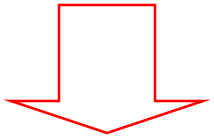
Example: 1st and 2nd passes

12	58	37	64	52	36	99	63	18	9	20	88	47



sort by rightmost digit

20	12	52	63	64	36	37	47	58	18	88	9	99



sort by leftmost digit

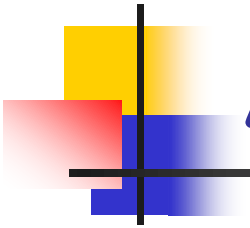
9	12	18	20	36	37	47	52	58	63	64	88	99

Radix sort and stability

- Radix sort works as long as the bucket sort stages are **stable** sorts
- Stable sort: in case of ties, relative order of elements are preserved in the resulting array
  - Suppose there are two elements whose first digit is the same; for example, 52 & 58
  - If 52 occurs before 58 in the array prior to the sorting stage, 52 should occur before 58 in the resulting array
- This way, the work carried out in the previous bucket sort stages is preserved

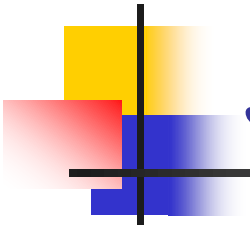
Time complexity

- If there is a fixed number p of bucket sort stages (six stages in the case where the maximum value is 999999), then radix sort is O(n)
  - There are p bucket sort stages, each taking O(n) time
- Strictly speaking, time complexity is O(pn), where p is the number of digits (note that $p = \log_{10} m$, where m is the maximum value in the list)



About Radix sort

- Note that only 10 buckets are needed regardless of number of stages since the buckets are reused at each stage
- Radix sort can apply to words
  - Set a limit to the number of letters in a word
  - Use 27 buckets (or more, depending on the letters/characters allowed), one for each letter plus a "blank" character
  - The word-length limit is exactly the number of bucket sort stages needed



Summary

- Bucket sort and Radix sort are O(n) algorithms only because we have imposed restrictions on the input list to be sorted
- Sorting, in general, can be done in O(n log n) time