Sorting Problems

Simple Version

- Given a sequence of integers in $A[n]$, output the non-decreasing list of $A$. 

Abstract Version

- In general, we could ask for sorting algorithms on any sequence with well-defined total orders.
- Recall total order: reflexive property, anti-symmetric property and transitive property. (on page 94).
- We have defined comparators that return comparison results in the abstract version.
Simple Version

- Given a sequence of integers in $A[n]$, output the non-decreasing list of $A$.
- For example, let $A = [9, 5, 7, 7, 1, 3, 1]$, output $[1, 1, 3, 5, 7, 7, 9]$. 

Sorting Problems

Simple Version

- Given a sequence of integers in $A[n]$, output the non-decreasing list of $A$.
- For example, let $A = [9, 5, 7, 7, 1, 3, 1]$, output $[1, 1, 3, 5, 7, 7, 9]$.

Abstract Version

- In general, we could ask for sorting algorithms on any sequence with well-defined total orders.
Sorting Problems

Simple Version

▶ Given a sequence of integers in $A[n]$, output the non-decreasing list of $A$.
▶ For example, let $A = [9, 5, 7, 7, 1, 3, 1]$, output $[1, 1, 3, 5, 7, 7, 9]$.

Abstract Version

▶ In general, we could ask for sorting algorithms on any sequence with well-defined total orders.
▶ Recall total order: reflexive property, anti-symmetric property and transitive property. (on page 94).
Sorting Problems

Simple Version

- Given a sequence of integers in $A[n]$, output the non-decreasing list of $A$.
- For example, let $A = [9, 5, 7, 7, 1, 3, 1]$, output $[1, 1, 3, 5, 7, 7, 9]$.

Abstract Version

- In general, we could ask for sorting algorithms on any sequence with well-defined total orders.
- Recall **total order**: reflexive property, anti-symmetric property and transitive property. (on page 94).
- We have defined comparators that return comparison results in the abstract version.
Lecture Plan

Week 9

- Monday (2/29): non-comparison-based sorting: bucket & radix sort (not directly applicable to the abstract version).
- Wednesday (3/2): comparison-based sorting: bubble, merge & quick sort (applicable to the abstract version).
- Friday (3/4): Lower-bound for comparison-based algorithms.
Bucket Sort

Setup

- Given a sequence $S$ of $n$ integers in the range $[0, N - 1]$. 

\[ \text{Bucket Sort: time } O(n + N), \text{ space } O(n + N). \]

Efficient when $N = O(n)$, i.e., time and space $O(n)$. 

Intuition

- Use one bucket for each possible value in the range $[0, N - 1]$. 
- Insert each key $k$ at the end of bucket (queue) $B[k]$. 
- Each queue at $B[k]$: linked-table-based implementation, insertion and removal $O(1)$. 
- Buckets $B[·]$: array-based implementation, space $O(N)$ and time $O(1)$. 

Buckets are naturally sorted. Output each bucket in order.
Bucket Sort

Setup

- Given a sequence $S$ of $n$ integers in the range $[0, N - 1]$.
- Bucket Sort: time $O(n + N)$, space $O(n + N)$.
Bucket Sort

Setup

- Given a sequence $S$ of $n$ integers in the range $[0, N - 1]$.
- Bucket Sort: time $O(n + N)$, space $O(n + N)$.
- Efficient when $N = O(n)$, i.e., time and space $O(n)$. 
Bucket Sort

Setup

- Given a sequence $S$ of $n$ integers in the range $[0, N - 1]$.
- Bucket Sort: time $O(n + N)$, space $O(n + N)$.
- Efficient when $N = O(n)$, i.e., time and space $O(n)$.

Intuition

- Use one bucket for each possible value in the range $[0, N - 1]$.
Bucket Sort

Setup

- Given a sequence $S$ of $n$ integers in the range $[0, N - 1]$.
- Bucket Sort: time $O(n + N)$, space $O(n + N)$.
- Efficient when $N = O(n)$, i.e., time and space $O(n)$.

Intuition

- Use one bucket for each possible value in the range $[0, N - 1]$.
- Insert each key $k$ at the end of bucket (queue) $B[k]$
  - Each queue at $B[k]$: linked-table-based implementation, insertion and removal $O(1)$.
  - Buckets $B[\cdot]$: array-based implementation, space $O(N)$ and time $O(1)$. 
Bucket Sort

Setup

- Given a sequence $S$ of $n$ integers in the range $[0, N - 1]$.
- Bucket Sort: time $O(n + N)$, space $O(n + N)$.
- Efficient when $N = O(n)$, i.e., time and space $O(n)$.

Intuition

- Use one bucket for each possible value in the range $[0, N - 1]$.
- Insert each key $k$ at the end of bucket (queue) $B[k]$
  - Each queue at $B[k]$: linked-table-based implementation, insertion and removal $O(1)$.
  - Buckets $B[.]$: array-based implementation, space $O(N)$ and time $O(1)$.
- Buckets are naturally sorted. Output each bucket in order.
Algorithm bucketSort(S)
Input: sequence $S$ with integer keys in the range $[0, N-1]$
Output: sequence $S$ with keys in the nondecreasing order

let $B$ be an array of $N$ sequences; initially empty

for each item $(k, x)$ in $S$ do
    remove it from $S$ and insert it at the end of $B[k]$
end for

for $i = 0$ to $N-1$ do
    for each item $(k, x)$ in $B[i]$ do
        Remove it from $B[i]$ and insert it at the end of $S$
    end for
end for

Time: $O(N + n)$; Space: $O(N + n)$.
Algorithm bucketSort(S)
Input: sequence S with integer keys in the range [0, N-1]
Output: sequence S with keys in the nondecreasing order
let B be an array of N sequences; initially empty
for each item (k, x) in S do
    remove it from S and insert it at the end of B[k]
end for
for i = 0 to N − 1 do
    for each item (k, x) in B[i] do do
        Remove it from B[i] and insert it at the end of S.
    end for
end for

▶ Time: $O(N + n)$; Space: $O(N + n)$. 
Bucket Sort: Example

- $A = [9, 5, 7, 7, 1, 3, 1]$, all items in the range of $[0, 9]$.
- Choose 10 buckets $B[0], \ldots, B[9] := [], [], [], [], [], [], [], [], [], []$. 

Output: $[1, 1, 3, 5, 7, 7, 9]$. 
Bucket Sort: Example

- $A = [9, 5, 7, 7, 1, 3, 1]$, all items in the range of $[0, 9]$.
- Choose 10 buckets $B[0], \ldots, B[9] := [], [], [], [], [], [], [], [], []$.

- $B[0 \ldots 9] = [], [], [], [], [], [], [], [], [9]$
- $B[0 \ldots 9] = [], [], [], [], [], [], [], [5], [], [], [], [9]$
- $B[0 \ldots 9] = [], [], [], [], [], [], [5], [], [7], [], [9]$
- $B[0 \ldots 9] = [], [], [], [], [], [5], [], [7, 7], [], [9]$
- $B[0 \ldots 9] = [1], [], [], [], [], [5], [], [7, 7], [], [9]$
- $B[0 \ldots 9] = [1], [], [3], [], [], [5], [], [7, 7], [], [9]$
- $B[0 \ldots 9] = [1, 1], [], [3], [], [], [5], [], [7, 7], [], [9]$
- Output: $[1, 1, 3, 5, 7, 7, 9]$. 


Stability of sorting algorithms

Stability
Let $S = \{(k_0, e_0), \cdots (k_{n-1}, e_{n-1})\}$ be the sequence to sort. The sorting algorithm is **stable** if

- For any two item $(k_i, e_i), (k_j, e_j)$ $(i < j)$, s.t., $k_i = k_j$ but $e_i \neq e_j$, we have $(k_i, e_i)$ precede $(k_j, e_j)$ in the output sequence.
Stability of sorting algorithms

Stability

Let $S = \{(k_0, e_0), \cdots (k_{n-1}, e_{n-1})\}$ be the sequence to sort. The sorting algorithm is **stable** if

- For any two item $(k_i, e_i), (k_j, e_j)$ ($i < j$), s.t., $k_i = k_j$ but $e_i \neq e_j$, we have $(k_i, e_i)$ precede $(k_j, e_j)$ in the output sequence.
- Important feature for the composition of sorting algorithm.
Stability of sorting algorithms

Stability

Let $S = \{(k_0, e_0), \cdots (k_{n-1}, e_{n-1})\}$ be the sequence to sort. The sorting algorithm is **stable** if

- For any two item $(k_i, e_i)$, $(k_j, e_j)$ ($i < j$), s.t., $k_i = k_j$ but $e_i \neq e_j$, we have $(k_i, e_i)$ precede $(k_j, e_j)$ in the output sequence.

- Important feature for the composition of sorting algorithm.

Bucket-sort is stable

- Bucket-sort is **stable** by enforcing every removal from the front and every insertion at the rear of sequences.
Radix-sort

Lexicographical (dictionary) order for multiple-component keys

- Assume the key $k$ has two components $(k, l)$.
- In the **lexicographical** order, $(k_1, l_1) < (k_2, l_2)$ if and only if

  - either $k_1 < k_2$, or
  - $k_1 = k_2$ and $l_1 < l_2$.

$k$: the first component; $l$: the second component.

Naturally, one can extend to multiple-component keys.

Examples

- The same way we compare integers; digit by digit.
- The same way we compare words; letter by letter.
Radix-sort

Lexicographical (dictionary) order for multiple-component keys

- Assume the key $k$ has two components $(k, l)$.
- In the **lexicographical** order, $(k_1, l_1) < (k_2, l_2)$ if and only if
  - either $k_1 < k_2$,
  - or $k_1 = k_2$ and $l_1 < l_2$.
- $k$: the first component; $l$: the second component.
Radix-sort

Lexicographical (dictionary) order for multiple-component keys

- Assume the key $k$ has two components $(k, l)$.
- In the **lexicographical** order, $(k_1, l_1) < (k_2, l_2)$ if and only if
  - either $k_1 < k_2$,
  - or $k_1 = k_2$ and $l_1 < l_2$.
- $k$: the first component; $l$: the second component.
- Naturally, one can extend to multiple-component keys.

Examples

- The same way we compare integers; digit by digit.
- The same way we compare words; letter by letter.
Radix-sort

Lexicographical (dictionary) order for multiple-component keys

- Assume the key $k$ has two components $(k, l)$.
- In the **lexicographical** order, $(k_1, l_1) < (k_2, l_2)$ if and only if
  - either $k_1 < k_2$,
  - or $k_1 = k_2$ and $l_1 < l_2$.
- $k$: the first component; $l$: the second component.
- Naturally, one can extend to multiple-component keys.

Examples

- The same way we compare integers; digit by digit.
Radix-sort

Lexicographical (dictionary) order for multiple-component keys

- Assume the key $k$ has two components $(k, l)$.
- In the **lexicographical** order, $(k_1, l_1) < (k_2, l_2)$ if and only if
  - either $k_1 < k_2$,
  - or $k_1 = k_2$ and $l_1 < l_2$.
  - $k$: the first component; $l$: the second component.
- Naturally, one can extend to multiple-component keys.

Examples

- The same way we compare integers; digit by digit.
- The same way we compare words; letter by letter.
Radix-Sort: 2-component keys

- Given a sequence $S[n]$ with 2-component keys, output in lexicographical order.
- e.g.,
  $S = \{(3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2)\}$. 

Radix-Sort

Radix-Sort: 2-component keys

- Given a sequence $S[n]$ with 2-component keys, output in lexicographical order.
- e.g.,
  $S = \{(3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2)\}$.
- desired output:
  
  $$S = \{(1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3)\}.$$
- just like $S = [12, 15, 17, 22, 23, 25, 32, 33]$. 
Radix-Sort: bucket-sort on the first component, and then the second

Bucket-sort on the first component

- Remember the bucket-sort is stable: we have,

\[ S_1 = \{(1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2)\} \].
Radix-Sort: bucket-sort on the first component, and then the second

Bucket-sort on the first component

- Remember the bucket-sort is **stable**: we have,

  \[ S_1 = \{(1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2)\} \].

Bucket-sort then on the second component

- Remember the bucket-sort is **stable**: we have,

  \[ S_{1,2} = \{(1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7)\} \].

- **Incorrect** order.
Radix-Sort: bucket-sort on the second component, and then the first

Bucket-sort on the second component

- Remember the bucket-sort is **stable**: we have,

\[ S_2 = \{(1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7)\} \].
Radix-Sort: bucket-sort on the second component, and then the first

Bucket-sort on the second component

- Remember the bucket-sort is **stable**: we have,

  $$S_2 = \{(1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7)\}.$$

Bucket-sort then on the first component

- Remember the bucket-sort is **stable**: we have,

  $$S_{2,1} = \{(1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3)\}.$$

- Correct order and can extend to multiple keys (in assignment 5).
Remarks

Correctness of 2-component key Radix sort

- By the stability of bucket sort, if two elements are equal in the second sort (by the first component), then their relative order in the starting sequence (which is sorted by the second component) is preserved.
Remarks

**Correctness of 2-component key Radix sort**

- By the stability of bucket sort, if two elements are equal in the second sort (by the first component), then their relative order in the starting sequence (which is sorted by the second component) is preserved.

**Radix sort for $n$ integer keys**

- To distinguish $n$ integer keys, these keys must have $\log(n)$ digits. Thus, one need to perform $\log(n)$ bucket sorts on each digit, which leads to $O(n \log(n))$. 